

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

THE DESIGN AND IMPLEMENTATION OF A REAL-TIME DISTRIBUTED APPLICATION EMULATOR

by

Timothy S. Drake

March 2001

Thesis Advisor:

Cynthia E. Irvine

Second Reader:

Jon Butler

Approved for public release; distribution is unlimited.

20010702 046

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washing Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE March 2001		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE THE DESIGN AND IMPLEMENATION OF A REAL-TIME DISTRIBUTED APPLICATION EMULATOR			5. FUNDING NUMBERS	
6. AUTHOR(S) Timothy S. Drake				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; Distribution is Unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) This thesis details the engineering, design and implementation of a real-time, distributed, application emulator system (AE system). The project had two main goals for the tool: emulation of real-time distributed systems, and as a programmable resource consumer. The AE system is currently being used in the HiPer-D test bed to activate a resource leveling tool that monitors several software components for real-time response. The AE system is highly flexible and can be used in the context of a variety of network topologies and system loading options. The results presented show that the AE system can also emulate distributed systems.				
14. SUBJECT TERMS Software Emulation, Real-Time Benchmarks,			15. NUMBER OF PAGES 138	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18 298-102

THIS PAGE INTENTIONALLY LEFT BLANK

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

THE DESIGN AND IMPLEMENTATION OF A REAL-TIME DISTRIBUTED APPLICATION
EMULATOR

Timothy S. Drake
B.S., Colorado State University, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

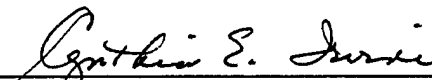
from the

NAVAL POSTGRADUATE SCHOOL
March 2001

Author:


Timothy S. Drake

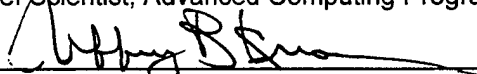
Approved by:


Cynthia Irvine, Thesis Advisor


Jon Butler, Second Reader



Michael W. Masters,
Chief Scientist, Advanced Computing Programs


Jeffrey B. Knorr, Chair

Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis details the engineering, design and implementation of a real-time, distributed, application emulator system (AE system). The project had two main goals for the tool: emulation of real-time distributed systems, and as a programmable resource consumer. The AE system is currently being used in the HiPer-D test bed to activate a resource leveling tool that monitors several software components for real-time response. The AE system is highly flexible and can be used in the context of a variety of network topologies and system loading options. The results presented show that the AE system also emulates distributed systems.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	MOTIVATION	2
B.	AE SYSTEM REQUIREMENTS	5
C.	ORGANIZATION.....	8
II.	RELATED WORK	9
A.	DYNBENCH.....	9
B.	CARFF'S EMULATOR.....	11
C.	PETRI NETS	14
D.	HARTSTONE BENCHMARK.....	15
E.	MSHN	17
III.	APPLICATION EMULATOR SYSTEM AND COMPONENTS	19
A.	INTRODUCTION.....	19
B.	PROJECT REQUIREMENTS	21
C.	SYSTEM DESIGN.....	22
1.	User Interface (UI).....	23
2.	AE Commands.....	28
3.	AE Unit.....	28
a)	CPU Loader	29
b)	Workload Module.....	35
c)	Networking	36
d)	AE Messages (Network Loading)	36
e)	Memory.....	39
f)	Message Table.....	40
g)	CPU Job Table.....	41
h)	Connection Table.....	42
i)	Network, Send and Receive	46
j)	Message Processing	54
k)	Controller.....	56
D.	SUMMARY.....	57
IV.	SYSTEM EMULATION AND EXPERIMENTATION RESULTS	59
A.	INTRODUCTION.....	59
B.	EADSIM.....	59
C.	SYSTEM EMULATION: THE THREE STEP PROCESS.....	61
1.	<i>Step One: Gather Resource Usage Data</i>	63
2.	<i>Step Two: Using Profile Data to Construct AE Commands</i>	64
3.	<i>Step Three: Running a System Emulation</i>	72
D.	EADSIM WRAPPER RESULTS	75
E.	SUMMARY.....	83
V.	DISCUSSION AND CONCLUSIONS	85
A.	LESSONS LEARNED	85
B.	FUTURE WORK.....	86
C.	COMPARISON WITH RELATED WORK.....	87

1.	DynBench	87
2.	Carff Emulator	88
3.	Petri Nets	89
4.	Hartstone.....	90
D.	CONCLUSION.....	92
APPENDIX A: AE COMMAND FILE FOR EADSIM EMULATION		95
APPENDIX B: AE COMMAND STRUCTURE		97
APPENDIX C: AUTOMATED EMULATION CONFIGURATION FILES.....		101
APPENDIX D: LIST OF ACRONYMS.....		103
APPENDIX E: DATA FROM A SERIES OF EADSIM EMULATIONS.....		105
LIST OF REFERENCES.....		115
INITIAL DISTRIBUTION LIST		119

LIST OF FIGURES

FIGURE 1 AE SYSTEM	23
FIGURE 2 AE UNIT BLOCK DIAGRAM.....	29
FIGURE 3 CPU LOADER FUNCTIONAL DIAGRAM.....	30
FIGURE 4 CPU LOADER TIME DIAGRAM.....	33
FIGURE 5 MESSAGE PATHS SUPPORTED	38
FIGURE 6 AE MESSAGE FIELDS.....	41
FIGURE 7 MESSAGE LAYOUT.....	48
FIGURE 8 EADSIM RUNTIME BLOCK DIAGRAM	61
FIGURE 9 EMULATION STEPS	62
FIGURE 10 TIME GRANULARITY EXAMPLE	67
FIGURE 11 AUTOMATED EMUATION DIAGRAM	74
FIGURE 12 EXPERIMENTAL RESULTS DIAGRAM (AVERAGES).....	82

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

TABLE 1 EADSIM RESOURCE USAGE DATA	76
TABLE 2 CPU RESOURCE USAGE DATA FOR EADSIM.....	77
TABLE 3 NETWORK USAGE DATA FROM EADSIM.....	77
TABLE 4 TARGET AND EMULATION RESULTS	80

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENT

Working on my Masters degree and thesis over two thousand miles away from the NPS campus has been a challenge. Without modern technology, I believe it would have been close to impossible.

I would like to thank my professors: Cynthia Irvine, Jon Butler and Debra Hensgen for their guidance and patience during the writing of this thesis. They all have helped a great deal, but special thanks goes to Professor Irvine who volunteered to take me as a student after Professor Hensgen left her position at NPS.

I have to add a special thanks to my wife, June and family. June helped many times on the issues related to writing a large document, but her best contribution was not letting me give up on my degree under the pressures of work and family.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

This thesis describes the design, implementation, and evaluation of a software tool that is capable of emulating real-time distributed applications. The tool is formally known as the Application Emulator (AE) system, and the primary goal of the project is to emulate Real-Time Distributed Systems (RTDS). This is achieved by providing software that can be easily configured to resemble a particular application, chosen from a wide range of real-time applications. The AE system is not meant to provide the functionality of real-time applications, but rather to imitate the resource usage patterns of such applications.

The AE was developed using an iterative process. Some iterations added functionality to the AE and allowed the AE system to emulate a wider range of RTDS. Other iterations concentrated on generalizing the design, emphasizing the concept of software reuse. These iterations tended to simplify the design. The final design has a scalable architecture that can be configured to emulate RTDS containing many components, each perhaps executing on a

different system, and each perhaps having real-time deadlines.

A. MOTIVATION

The research performed for this thesis contributes substantially to the Naval Surface Warfare Center's (NSWC) High Performance Distributed (HiPer-D) computing project. The HiPer-D project is currently developing a prototype, next-generation high performance Anti Air Warfare Weapon Control System (WCS). The project is focused on determining whether Commercial Off The Shelf (COTS) systems can meet the real-time, scalability and fault tolerance requirements of such applications. If successful, the move to COTS will offer several advantages over dedicated systems including:

- lower software and hardware costs,
- higher performance (faster computationally) computers in terms of processing power,
- reduced hardware upgrade times, and
- user familiarity with interfaces and components.

The move to COTS to support these applications represents a major paradigm shift. The currently fielded set of WCS applications is supported by special-purpose, dedicated hardware (i.e., computers). The communication

between components is supported by point-to-point dedicated connections. The set of fielded applications comprises a large and complex entity. The prototype, currently being developed and analyzed in the HiPer-D laboratory at NSWC, while also large, does not encompass the entire functionality of the fielded application. Adding the full functionality to the existing laboratory code would require a substantial investment. Therefore, an AE system that can be configured to accurately emulate the software components that are part of the fielded system but are not part of the prototype would aid in analyzing the suitability of COTS for these applications at a fraction of the cost. An AE used in this context must accurately mimic the loads that would be placed on the computing and communication resources by the missing components.

Part of the HiPer-D mission is to determine whether the next generation WCS can meet its requirements if implemented using COTS components. Furthermore, if the current COTS systems do not meet the needs of such applications, the HiPer-D project must identify the areas of today's technology that fail to provide such support. Finally, when such areas are identified, the project may also suggest avenues for new COTS technologies that will better meet Navy's application requirements.

This motivation helps explain the basic requirements of the AE project. An application observed from an external vantage point has a CPU usage pattern (or profile), a network usage pattern, and a memory usage pattern, in addition to usage patterns for some less obvious resources such as file server access. The main goal of this research, therefore, is to design and implement software that can be easily configured to accurately imitate the resource usage patterns of WCS software. Obviously, the algorithms that will be used by next-generation WCS may be different from those used in today's systems. Therefore, the AE must be able to imitate not only the usage pattern of today's WCS applications, but tomorrow's as well.

The algorithms used in WCS applications will likely change over time, yet based upon existing WCS applications [T3], it is clear that any algorithm that performs weapon control will fall into a class of applications known as periodic, real-time applications. The main characteristics of this class of applications are that they repeatedly receive sensor or pre-processed information, execute one or several filtering-type algorithms, and report an answer before a deadline. Such applications are both CPU and network-intensive. Therefore, the design for this AE project has focused on two main areas: the ability to

replicate CPU usage patterns and the ability to replicate network usage patterns. Additionally, the temporal relationship between these two uses must also be replicated. As part of this thesis, we discuss how this design may be expanded in the future to include replicating usage patterns for other resources.

Using the AE system, together with the components of the next-generation WCS that have already been implemented, will provide a higher level of confidence concerning conclusions about the ability of COTS to support the next-generation WCS. Without the AE system, or alternatively the costly implementation of the rest of the functionality of the application, the adequacy, strengths, and weaknesses of the COTS system might be unknown. For these reasons, the AE is an important tool for the HiPer-D project.

B. AE SYSTEM REQUIREMENTS

The AE system was designed and built to emulate RTDS. As such, it has the following high-level requirements:

- Its architecture must be both distributed and scaleable.
- It must be written in a language that is portable, supports multiple threads. It must be designed to reduce life cycle costs.

- It must be capable of being configured to emulate real-time applications that have periodic deadlines.
- It must be possible to determine whether performance requirements, such as deadlines, are met.
- It must produce similar resource loads when run repeatedly with the same parameters. In particular, it must produce repeatable CPU and network usage patterns.

Compliance with the above requirements is discussed below.

The AE system must be able to mimic applications comprised of many components, although the separate components may execute on different systems. In other words, the AE must consist of components that can be replicated, individually configured, and distributed. The current requirement is that the AE must be able to mimic the operation of an application consisting of at least twenty different communicating components, which may be running on any number of systems that support the defacto LAN standard, TCP/IP.

The second requirement is that the emulator must include a wide range of features. Although many other modern languages meet the language requirements of the AE project, the decision to develop the AE in Ada95 was largely due to a previous Navy requirement that stipulated the programming language that must be used for Naval real-time applications.

The third requirement deals with the real-time characteristics of the AE project. For the purpose of this thesis, we will use the two terms hard real-time and soft real-time as they are commonly used in the literature. A component with *hard real-time* deadlines must complete its periodic work before the deadline for each period in order to satisfy system requirements. *Soft real-time* applications meet their requirements if the statistical mean of the sample distribution of response times satisfies the deadline constraints [Lui73]. For the purpose of this thesis's AE, hard real-time constraints were interpreted to mean that missed deadlines must be reported. In many applications with real-time periodic deadlines, the deadline of the previous period is also the start of the next time period [Hart89].

In order to meet the last requirement described above, the AE must have a repeatable way to apply a load to the network and the CPU, as well as to other resources. The networking load requirement requires that the AE must be configurable to allow the components to send and receive messages to one another in such a way that the dynamic message-passing topology can easily be specified. The CPU loading specification should ideally be independent of the speed and instruction set of the processor.

C. ORGANIZATION

The remainder of this thesis is organized as follows: Chapter II contains an overview of related work. Chapter III discusses the details of the AE and its components. It covers the desired features of the AE, first at a system level, and second it includes a detailed discussion of the components in an AE unit. Chapter IV provides an analysis of the AE system including an overview of the emulation steps. The chapter includes a section on the application being emulated (EADSIM). The chapter finishes with a section on results optioned and an analysis of the AE system. Chapter V concludes the thesis by discussing some lessons learned while developing the AE, by suggesting future work for the AE, and, by contrasting the AE with related tools for real-time system emulation.

II. RELATED WORK

This chapter describes several simulation and emulation tools that are closely related to the AE system. A comparison between the tools discussed here and the AE is presented in Chapter V after the reader has had an opportunity to read Chapters III and IV and has a clearer understanding of the AE system.

A. DYNBENCH

DynBench [WELC98] is a benchmark suite that was designed to emulate a portion of the prototype next generation Anti-Air Warfare, or, as it is better known, HiPer-D [T3]. Therefore, DynBench is a real-time distributed system with Quality of Service (QoS) requirements. Just like HiPer-D, DynBench is a system that allows certain software components of its distributed system to be relocated during operation. Relocating a software component means moving it from one computer to another computer. Relocation of runtime components is a feature that requires an outside action (i.e., some other software component to act on resource data, and to actually kill and

restart the DynBench components being moved) and is normally in response to problems related to loading or faults¹. For example, any or all the components that make up a DynBench run-time system can be relocated while DynBench is operating.

DynBench maintains Quality of Service (QoS) data and provides an Application Programming Interface (API) to allow a Resource Management System (RMS) or some other process access to the QoS data. This data can be used for intelligent run-time adaptation.

The primary load on a DynBench system (and the tactical systems that it emulates) is a function of the number of objects, or radar tracks in the radar view. Radar tracks have various attributes such as speed, heading, and identification, i.e., friend or foe. DynBench provides two methods for changing the track load. The first is based on time, where the number of tracks in the system is increased or decreased as a function of time. The second allows the number of tracks to be instantaneously changed by an issuing interactive command.

¹ A fault can be a computer system that fails or a network segment that also fails and causes a computer system to be isolated.

Other characteristics of radar tracks that DynBench must account for include initialization (where they start) and heading (where they are going). These characteristics are important when studying a tactical system, but are much less important if one is using DynBench as a loading tool and not as a tactical system.

The DynBench benchmark suite uses simulated data in its messages and processes this data with accepted algorithms. In contrast, most real-time benchmarks use a synthetic workload² created by calling existing CPU loading benchmarks (e.g., Whetstone) for workloads. This places DynBench somewhere between a general-purpose benchmark and an actual system.

B. CARFF'S EMULATOR

Paul Carff [CARF99] performed research aimed at determining how much data are needed about an application's run time resource usage in order to predict how it will perform on different platforms (i.e., different processor, memory and operating system configurations). Collecting the

² A synthetic workload is a nonreal-world program that usually exercises one aspect of a system. An example is the Whetstone benchmark that performs operations on floating point numbers.

right amount of resource information is a difficult problem. If not enough information is obtained, then the resulting, prediction may be incorrect by more than an acceptable percentage. Large amounts of data can put a strain on the run-time performance of the system under study and increase the problem of managing and processing the data. The optimum level of data collection should allow prediction to occur within a certain level of accuracy (e.g., plus or minus 10%) at some level of consistency (e.g., 90% of the time).

The application Carff utilized was a distributed message passing application that allows for a configurable number of components. Systems such as MSHN (section II.E) are designed to use runtime data similar to that collected by Carff to predict resource requirements, and, when possible, completion time estimates for applications. MSHN was intended to use this information to decide where an application should run, while accounting for many factors including:

- the deadline of an application,
- the current state of the system, and
- other pending jobs.

Carff developed an emulator to validate his thesis. His emulator, developed in Java, contains four modules, each of which is always executed within its own thread. A brief description of each module is given below.

- Main. This module receives the information needed to configure the other three modules. It also must wait until the other modules complete before exiting.
- Calculation. This module performs the CPU loading by multiplying two 100×100 matrices. It instructs the sending modules to send messages based on either a fixed interval or a statistical distribution on the progress made in finishing the multiplication.
- Sending. Each instance of Carff's emulator has a send thread for every other emulator in the test. If there are n (e.g., $n=5$) copies of the emulator running then each will have $(n-1)$ copies of the send module (e.g., 4). Each module sends messages to only one remote emulator.
- Receive. This module is very similar to the Send module. Each emulator will have $n-1$ copies of the receive modules running. Each receive module is dedicated to receiving from a single remote sender.

The emulator runs until all threads have completed their work. This work includes reception of all messages and, of course, finishing work that emulates CPU loading. Only when the calculation, and receive and send threads have completed will the main module terminate.

C. PETRI NETS

Petri nets can be used as a tool for the indirect study of a system [PETE81]. The first step to utilizing Petri nets is to create a model of an existing system by incorporating all important features of that system into the model. Then, the model, which is mathematical in nature, can be analyzed to learn more about the actual system.

Many different systems can benefit from this indirect method of study. There are many cases when the Petri net method can work better than studying the actual system. For example, astronomy (where the times involved in the actual system are too great) and sociology (where studies might cause ethical problems) are eminently suitable for studies via Petri nets [PETE81]. The DynBench benchmark suite is an example of a model representing a system that is not readily available for experimentation since the software is considered proprietary and sensitive (for national security) in nature. The operating signature (resource utilization of the COTS components) does not have the same level of classification and can be used in the model.

Well-constructed models are necessary for useful Petri nets. If a model is poorly constructed then the resulting conclusions can and probably will be incorrect. If a model is well constructed and correctly represents the major features of the real world system, then the Petri net study can yield usable results. The interaction between the components (which may be large complex systems themselves) must be preserved in the model. Each component of the system has a separate behavior as well as an interaction with the other components. That behavior may change over time and/or events (the current state).

D. HARTSTONE BENCHMARK

Hartstone is a real-time benchmark suite. It was initially proposed as a specification [HART89] for a benchmark suite, but later was developed into a working tool. The premise of the Hartstone benchmark is that developers can prototype a proposed real-time system, and then execute that prototype on the intended computer hardware. This allows the designers to quickly see how the actual system will perform within some margin of error when fielded. Before Hartstone, developers would either wait

until the system was built or would have developed a prototype themselves in order to test system response.

The Hartstone Benchmark provides a series of tests to conduct on a real-time system [HART89]. The five tests defined by Hartstone [HART89] are given below:

- PH Series: Periodic Tasks, Harmonic Frequencies
- PN Series: Periodic Tasks, Non-Harmonic Frequencies
- AH Series: PH Series with Aperiodic Processing Added
- SH Series: PH Series with Synchronization
- SA Series: PH Series with Aperiodic Processing and Synchronization

Harmonic frequencies means that all the periodic tasks have a common base frequency. The task with the shortest time period, operates at the base frequency. All the other task's periods are some multiple of the base frequency. For example, if a system includes three tasks, and they have the following periods: Task One is one second, Task Two is two seconds and Task Three is four seconds, then these tasks are harmonic with a base frequency of one second (i.e., Task One is the task that runs at the base frequency). When the tasks are synchronized and harmonic they all start at the same time, so every fourth second all three tasks will start an execution cycle.

A test utilizing the Hartstone benchmark continues until the system misses a hard real-time deadline. If, during the first run of the system all hard deadlines are met, then one part of the system is changed to make meeting the deadlines during the next run more difficult. This continues until the system fails to meet a hard deadline. The Hartstone benchmark is intended to measure the breakdown point of a real-time system [HART92]. Hartstone benchmark results allow a real-time system designer to know before software development if the end product could operate as specified in terms of real-time response.

E. MSHN

The Management System for Heterogeneous Networks (MSHN) was a research project to develop a Resource Management System (RMS). One goal of a RMS system is to make a set of distributed computational resources (heterogeneous in MSHN's case) look and act like one virtual machine [HENS99]. Distributed Operating Systems are also tools that attempt to make a set of networked machines look like one virtual system. One major difference between a RMS and a distributed operating system is that the RMS does not manage system resources. That task belongs to each local operating

system. Thus, an RMS provides a mechanism for intelligently assigning applications to a computing system selected from set of computing resources.

One objective of MSHN was to attempt to meet QoS requirements by supporting adaptive applications [HENS99]. Adaptive applications allow for different levels of fidelity in the output. For example, directions to a local airport can be delivered as a color map, a black and white map, or, in the simplest case as an ASCII description. If the completion time is critical, then meeting that deadline (in this case, perhaps catching a plane) is the primary concern and the quality (in terms of display) of the result is secondary. The intended MSHN goal was to meet deadlines with the best quality results, but if available resource levels would not permit the deadline to be met, then to adapt one or more of the applications so that the deadline could be realized.

The adaptable features described for MSHN appear promising for future applications. The idea of applications that dynamically adapt themselves through tools like MSHN, allowing several competing applications to all meet their deadlines, would be a major advancement over today's static software systems.

III. APPLICATION EMULATOR SYSTEM AND COMPONENTS

Chapter III has the following organization. The introduction contains a high level overview of the AE system. Section B, contains background information and the constraints on the AE project. The chapter finishes with a section giving the functional block diagram of an AE unit. This section also contains a detailed description of the components that make up an AE unit.

A. INTRODUCTION

In developing the AE, the primary goal was to keep the components of the AE unit fairly generic and flexible. Although this may seem to constrain the capability of the AE system, it will be shown that the AE system can emulate a large Real Time Distributed System (RTDS). Large system emulation is possible because each AE unit can mimic most real-time components and large systems can be constructed by interconnecting them in a wide range of topologies.

In the HiPer-D program one of the early needs for an emulation or simulation tool was to place additional load on partial implementations of large real-time systems. In this role, the AE system would emulate missing or unfinished

components that are part of a larger system. Its role would be to emulate the missing component's resource usage profile and thereby create the illusion of having the component present. The resulting system, consisting of actual code and the AE system, can then be studied. The ease of modifying the AE system usage profile allows a wide range of tests to be conducted much like a Monte Carlo simulation. In this role, the AE system is utilized as a tool for quickly prototyping real-time components.

The CPU cycles consumed by an AE unit can be divided into two areas, overhead and emulation. The overhead CPU usage is introduced by the AE as it executes an emulation and is a necessary aspect of any emulation package. The AE system, at a system and component level was carefully designed and coded to minimize CPU processing while maintaining a acceptable level of functionality. The CPU emulation, which is central to application emulation, is accomplished by placing a synthetic workload on a system.

Each AE unit provides the ability to emulate the resource utilization of the following system resources: CPU, network and memory. The rest of this chapter will cover the detailed requirements and the design of the Application Emulator. Further, it will show how the design and implementation meet the requirements.

B. PROJECT REQUIREMENTS

The initial goal of the AE project was to provide an application emulator that could mimic the resource utilization of existing or planned real-time applications. A working group from the HiPer-D project formulated the following high-level requirements for the AE development program:

- It must be written in a high-level language such as Ada.
- It must allow configuration via a centralized controlling unit.
- It must include a realistic CPU loading capability.
- It must support emulation of periodic real-time processes with deadlines.
- It must include emulation for transient periodic real-time processes.
- It must support a complex message passing capability that includes performance metrics.
- It must be written such that it can operate in a heterogeneous environment.
- Each experiment must be repeatable.
- The metrics provided must be useful for performance (i.e., loading) tuning.
- In general, it must provide the ability to simulate processing and communication workloads on multi-computer networks.

C. SYSTEM DESIGN

This section gives the reader an overview of the AE system's design, and insight into how that design meets the requirements of the AE project. First, we will start by looking at how the AE system operates, and then dissect an AE unit, with a detailed description of its components. Figure 1 illustrates a prototypical AE system: one or more AE units and a User Interface (UI). The most important component of an AE system is the AE unit, which is the emulator engine that emulates real-time software components.). This diagram is similar to the system used for experimentation to be discussed in Chapter IV.

Each AE unit in Figure 1 can be located on any computer and operates independently of the other AE units, so all three AE units in Figure 1 could be located on the same computer or three different ones. To change the experiment and move an AE unit to a different computer only requires changing the platform where each AE unit is started. It is worth noting that the example in Figure 1 does not account for network traffic between the AE units. In a typical RTDS, network traffic would be present.

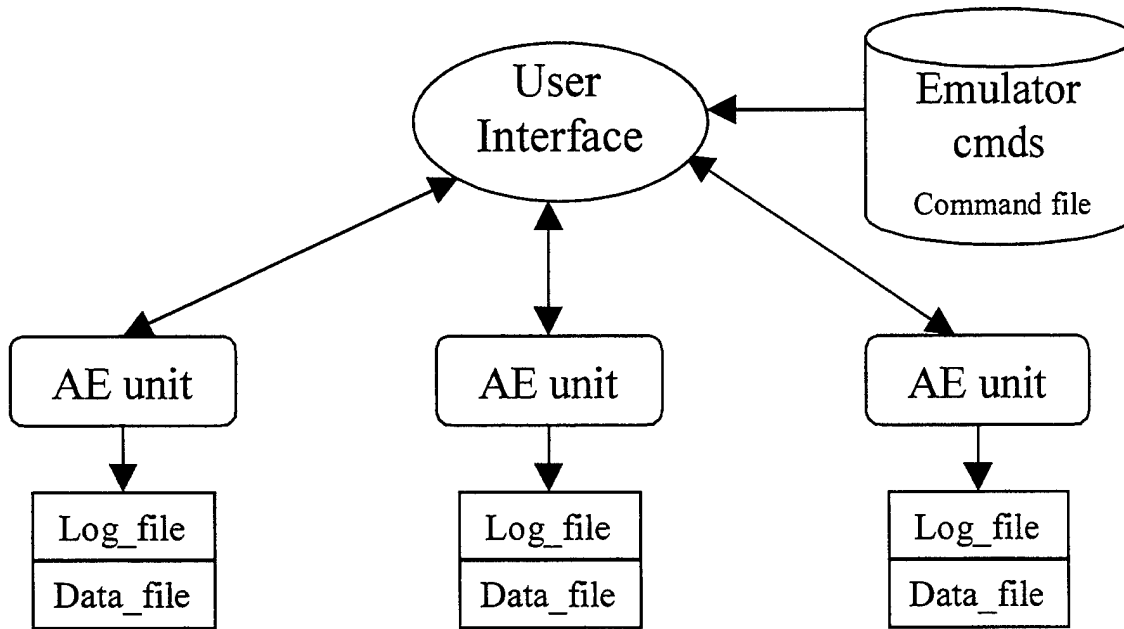


Figure 1 AE System

The rest of this chapter describes how the AE system operates.

1. User Interface (UI)

Emulation using the Application Emulator (AE) system consists of the objects shown in Figure 1: Command file, User Interface (UI) and one or more AE units. The UI is a multi-threaded application whose main task is to control the execution of a group of AE units operating as a RTDS. The command file contains a list of commands that give each AE unit its resource usage profile. The command file drives

the emulation by providing the resource usage information and, in addition, provides a method for specifying precise timing of resource utilization.

As shown in Figure 1, the UI plays a major role in the execution of the AE system. The procedure for starting all the AE units and the UI for an experiment is described below. After an experiment starts, the UI reads and processes the command file. Each command in the command file contains a field, which specifies a particular AE unit by name. These commands contain the information that controls what resource and the amount of each resource each AE unit will use. Different command files allow the AE system to emulate an entirely different system.

The UI satisfies several of the high-level requirements for the project. It provides the centralized control and, by employing the command file, provides for repeatable experiments. The UI also allows the AE system to be scaleable (easily supporting many AE units in an experiment), while also supporting the distributed architecture requirement.

The startup procedure for the AE system can aid in understanding how it operates. Each AE unit is known by its name, a string of up to 14 characters. Use of names to identify and establish connections in the network, allows an

AE unit to be executed on any computer system on the Local Area Network (LAN). Starting the applications manually (the UI and the AE units) on several different computers is a multi-step process. To save time and reduce mistakes, an automated startup tool that greatly simplified the otherwise labor-intensive task of startup was borrowed from the DynBench project. The startup task involves one UI and one or more AE units.

There are two initialization (i.e., command line) parameters of interest for the UI. The first specifies the mode of operation: interactive or batch. The default mode of operation is batch in which commands are processed from a file, but for testing and flexibility, an interactive mode is available. The interactive mode includes a tool that can help a user to construct the lengthy AE commands. The second is an optional parameter that specifies the number of AE units that will be part of the experiment. This parameter instructs the UI to wait until that number of AE units has connected to the UI (via TCP/IP) before processing commands. After the last AE unit creates a connection with the UI, it starts processing the command file. The UI also records the start time, which is used for Quality of Service (QoS) data and for commands that require timing (described below).

There are four types of commands and all of them support an optional time field parameter. This option is available in both modes of operation (interactive and batch) but would be difficult to use effectively in interactive mode, since interactively constructed AE commands are likely to execute late. Part of the process of parsing commands includes checking for the time parameter. If the time field is present, then the UI must determine if the command is to be executed immediately or later. This calculation is made by comparing the elapsed time and the command's time field parameter. Elapsed time is defined as the current time minus the start time. If the current command needs to wait, then the UI suspends execution until the time field and the elapsed time are the same; at this point it sends the command to the intended AE unit. Because the command file is processed from top to bottom, all commands that follow one with the time parameter specified must also wait until it is processed.

An AE unit supports several command line parameters, but, for normal operation, only two are significant. The first one specifies the AE unit's name. The second one is the name of the host where the UI is executing. This information allows this AE unit to make a network connection with the UI. The mapping of an AE unit name to IP address

takes place at run time and as stated above allows AE units to be executed on different computers for different experiments while using the same command file. The name information is stored in the connection table, which is described in detail in Section III.C.3.h). The connection table is replicated on the UI and at each AE unit. This feature allows an AE unit to be easily extended so that it can be moved during runtime (this extension is not yet implemented). The table includes enough information (i.e., AE unit names and IP addresses) so that each AE unit can create a network connection with any other operating AE unit.

In summary, the UI plays a major role in the overall operation of an experiment. It can be used to synchronize the startup process, and, because it records the start time, it also allows for the precise timing of individual commands. The UI's architecture and implementation allow centralized control. Centralized control and the naming feature allows the individual AE units to be located anywhere. The UI ends an experiment when it encounters the "stop all" command. It forwards the command to all the participating AE units, informing them to perform a normal shutdown.

2. AE Commands

The AE supports four types of commands. Each command in the command file is one of the following types:

- CPU command,
- network command,
- memory command or
- control.

The first three command types are used to specify resource loading for a particular resource. The command structure developed for the AE system is shown in Appendix B and an example is included in Appendix A. The control command type is used for shutting down the system after an experiment or test has completed.

3. AE Unit

Figure 2 depicts all of the major internal modules of an AE unit and most of the interactions between the modules. All the shaded objects represent a process thread. As shown, the AE is a multi-threaded, complex application. Its components include: message table, connection table, network modules (i.e., send receive and a general networking module), CPU job table, monitoring, message processing, CPU

loaders, benchmarks, controller and the memory loader. The rest of this section describes each AE component.

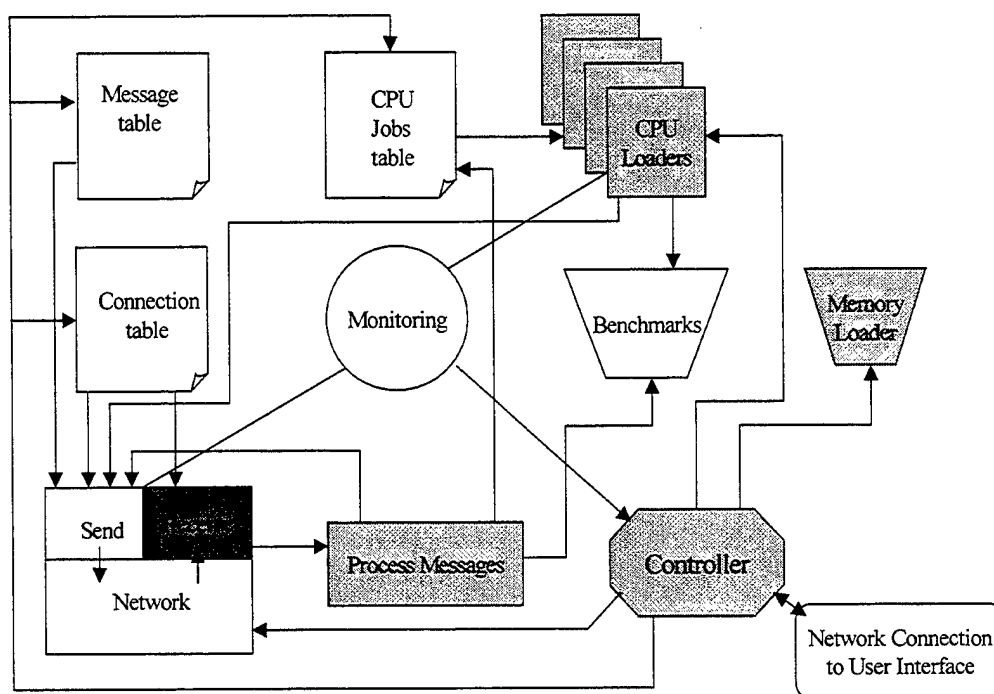


Figure 2 AE Unit Block Diagram

a) CPU Loader

Figure 3 is a diagram showing how a CPU loader operates. It is important to note that, for real-time processes, the main loop will operate forever (until it is stopped) and not a specified number of times. The CPU Loader along with the workload module emulates the CPU workload of periodic, aperiodic, or transient periodic real-

time processes. Each AE unit has the ability to support multiple, concurrent executing CPU loaders.

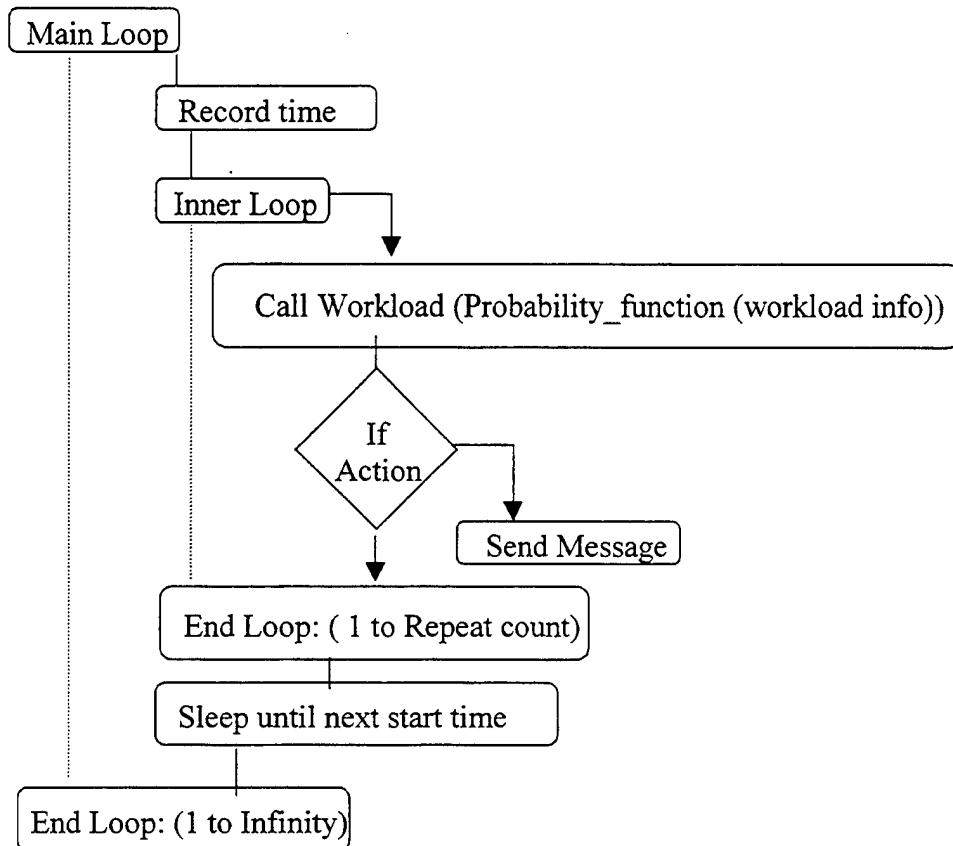


Figure 3 CPU Loader Functional Diagram

The CPU loader module has several features that need explanation. The call to the workload module causes CPU emulation to be performed. The item labeled "action" drives the network emulation capability. The term "action" and how it applies to network emulation is described below.

The last feature is the repeat count; this feature allows each instance of a CPU loader module to have specified both a period and a repeat count. For example, if a CPU loader was defined with a period of two seconds, a repeat count of four and an action to send a message. It would operate as follows, the main loop would start every two seconds. The inner loop would iterate four times for each main loop execution. Each time the inner loop executes it would call the workload module and, because an action is defined, it would also send a network message. Therefore, every two seconds the loader would call the workload module four times while also sending four messages. The sequence described above is also shown in the timing diagram shown in Figure 4. The term *action* as it is used in this thesis, is defined as, "linking CPU processing to the loading of other resources", such as sending a message after completing a defined workload. In the general case, real-time components complete the same task repeatedly in a periodic (e.g., every second) nature. The repeat parameter allows a periodic CPU Loader's period to be divided into segments so that an *action* can occur several times in a single period (as diagramed in Figure 3). The parameters listed below are configurable at CPU loader initialization time:

- real-time period (in milliseconds),
- benchmark (Whetstone or Dhrystone),
- workload data: average, and distribution parameters (e.g., distribution parameters can be mean and variance),
- action and action probability (for example the action is taken 60% of the time), and
- repeat value (allows actions to occur several times in a single period).

Each CPU loader task maintains the following Quality of Service (QoS) data:

- deadlines: missed and met, and
- message end-to-end timing information.

The CPU loader modules can operate in either a periodic or an aperiodic manner. Figure 4 illustrates a loader module as seen by someone tracing its execution through a single period.

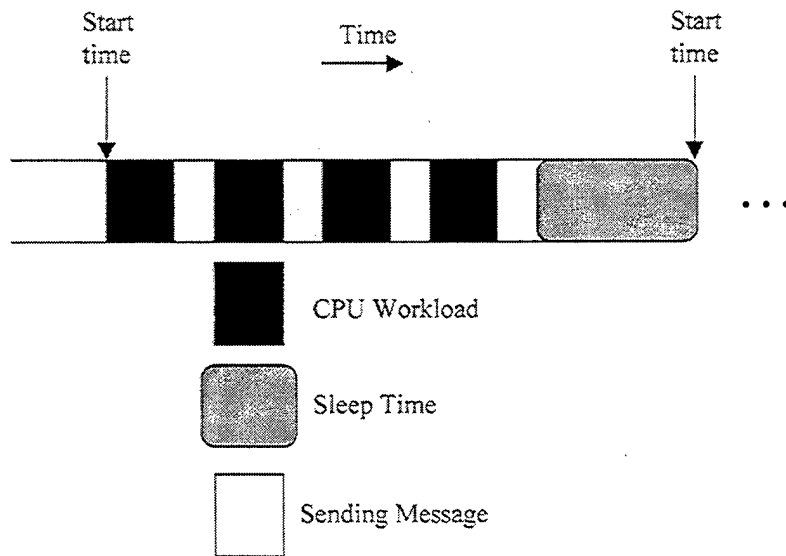


Figure 4 CPU Loader Time Diagram

Figure 4 shows a periodic CPU loader operating with a repeat value of four and an action to send a message (note, that in this example the probability to transmit a message is 100%). Remember that real-time applications operate in a periodic fashion, wake up, process, sleep, wake up, process sleep, etc. Here, periodic means that the start times are uniformly spaced in time. The above diagram is a snapshot of such a process, and moving ahead, or back in time will produce a similar diagram with evenly spaced start times. The diagram shows the loader first simulating CPU usage followed by the transmission of a message (through an action). The above sequence is executed four times in the

diagram, at which time the loader has finished its CPU and network emulation for the execution cycle. It will then suspend execution until the next cycle is due to start.

When a CPU Loader starts a new execution cycle, it first calculates the CPU workload using the average and statistical distribution data. The data used in the calculation are contained in the AE command that describes the CPU loader. Workloads can be described as normal, uniform or exponential statistical distributions. Next, a time stamp is recorded to allow for QoS measurements. The workload information is then sent to the benchmark module to emulate CPU loading. As an example, this module might call the Whetstone Benchmark to simulate CPU loading, or workload as it is referenced in this thesis. Each action has an associated probability (0% to 100%) that is checked before the action is executed. So, if an action is defined and if the probability test passes, a call to the "network send" (sending a message is the only implemented action) module is made with the information needed to construct the size and type message being sent. If a repeat value is set, then the process described above is repeated for the specified number of times. Finally, the next start time is calculated. If the next scheduled start time has passed, then a deadline was missed. The loader records the event (deadline missed),

and starts the next iteration. If the deadline was met, the loader will issue a sleep command to consume the remaining time.

b) Workload Module

The main function of the workload module is to emulate an application's CPU resource utilization. To accomplish this task the AE uses a list of commonly available benchmark programs, which provide a synthetic workload. The list of benchmarks supported includes a small Whetstone [CURN76] and a Dhrystone [DHR84] benchmark. These two benchmarks were selected because they represent computationally intensive workloads and the class of software being emulated (real-time distributed) normally can be characterized as having the same characteristics. For completeness and flexibility, the design and implementation of the AE allows additional benchmarks to be easily added to the existing set.

c) Networking

The AE system has been developed to emulate existing and/or planned RTDS. In that environment, some applications only process messages; their workload is a function of the number and type of messages that they receive. A message received by an AE unit can contain workload information. Details of message content and how messages are processed by an AE unit are fully explained in a subsequent section (Message Processing III.C.3.j).

d) AE Messages (Network Loading)

An AE message consists of several data fields, name fields (i.e., AE names), and QoS fields which contain timing information. The name fields contain the originator and all the receivers of that message. The names define the arcs that a message takes through an AE topology and determine what communication connections are required to support that message. Four different message path types are supported by the AE project: simple, fan out, pipeline and circular pipeline. They are illustrated in Figure 5 and

were selected because they encompass most of the communication functionality found in modern large real-time distributed systems today. The more complicated message types (pipeline, fan out and circular pipeline) can have up to five receivers. Five was selected because it was large enough to allow the AE to emulate the most complex message passing used in the HiPer-D prototype. A larger number was not selected because each message transmitted between AE units carries the entire data structure required to support all the features of the networking subsystem. The overhead of supporting up to five receivers, adds 300 bytes to each message. A larger number would have increased the overhead of the AE.

All commands (from the command file) are processed through the UI and then passed to an AE unit as described in the UI Section III.C.1. Message commands require an additional parsing step by the UI to decode the message type and to extract all sender and receiver information. The sender and receiver information is then sent to the affected AE units to inform them of the required network connections. The following parameters are set when defining a message (when creating an AE network command):

- message type: simple, fan out, pipeline or circular pipeline

- message information: size and message size statistical distribution parameters (e.g., size mean and variance)
- protocol (UDP, TCP) and port number
- the number of receivers and their names
- unique workload information for each receiver of this message including which benchmark to use for CPU emulation.

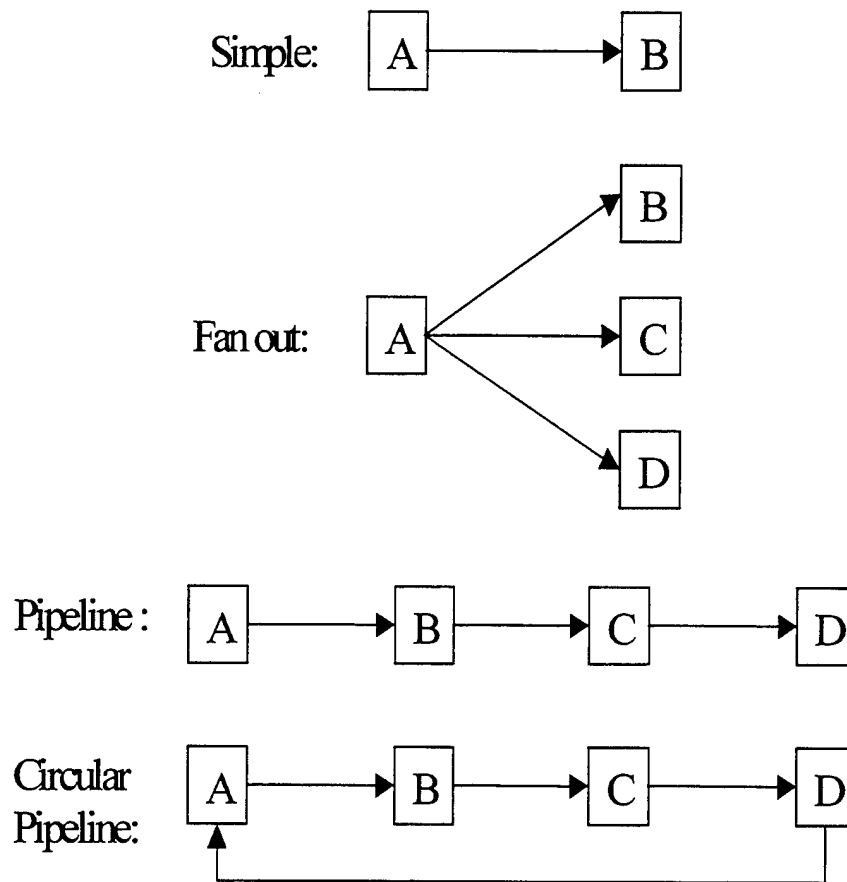


Figure 5 Message Paths Supported

e) Memory

The memory emulation capability provided by the memory module uses a rudimentary approach. The minimum memory consumed by an AE unit is approximately one megabyte of memory. Memory usage is emulated by allowing an AE unit to expand its total memory usage. There are two commands for memory emulation: one that adds to the current size of an AE emulator and an other that decreases the emulator's size (this command must be preceded by a command that increases the size). Reduction in memory size cannot go below the actual size needed for the AE unit itself. For example, if a particular AE unit was emulating an application that has a run-time size of 3.5 megabytes, then the AE would need to add 2.5 megabytes to its memory allocation to use the same amount of memory. The AE emulates the application memory footprint and not its memory access.

Memory is merely allocated is not used or accessed in any manner by an AE unit. Normally applications allocate memory for a reason, and they normally use that memory for code or data.

f) Message Table

AE units store network messages in the message table (see Figure 2). When the UI processes a network message, the UI sends a copy of that message to the message originator (i.e., the AE unit that will initiate the sending of that message). Remember that an AE system can, in theory, support a very large number of messages, and the discussion below describes a single message. All messages are static in that they always start from the same AE unit and traverse the same ordered set of AE units. The originating AE unit receives a copy of the message from the UI via the controller as shown in Figure 2. That AE unit then inserts that message into its message table. The actual transmission of the message requires the *send module* (see Figure 2) to obtain a copy of the message from the message table.

Figure 6 shows the data structure common to all AE messages. At the top of the diagram are the fields that define the number of receivers and the type of the message (see Figure 5 for a full list of types). This is followed by the workload data structure. The workload data structure contains the topology information (contained in the AE Name field) and the workload information for each receiver of the

message. The Action data structure at the bottom of Figure 6 is optional. That field defines an action (where "no action" is a valid option) and the probability associated with actually executing the action.

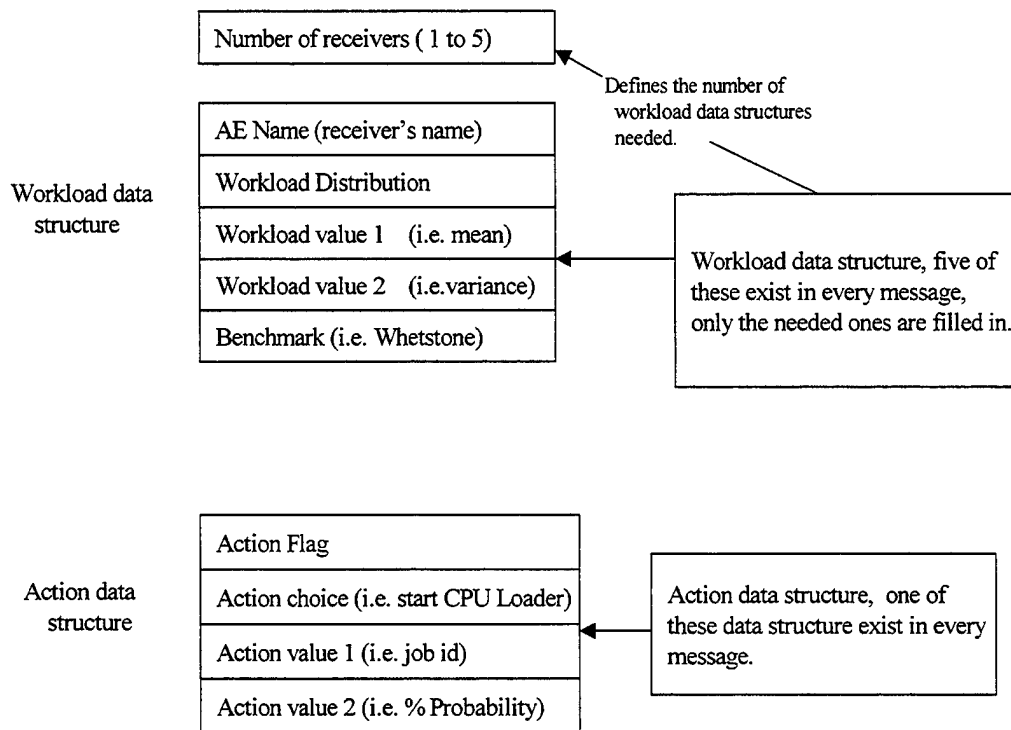


Figure 6 AE Message Fields

g) CPU Job Table

Periodic and aperiodic tasks are emulated using CPU loader jobs. CPU commands take the same path as network commands, proceeding from the command file, through the UI

to the appropriate AE unit. Most CPU commands become CPU loader jobs when they are received, but some are started and/or stopped by events (i.e., transient periodic processes). The job table is where event processing obtains the parameters to configure and start a CPU Loader process.

h) Connection Table

Each AE unit maintains its list of active network connections with other AE units in the connection table. New connections between AE units are created only when required by a network message command. For example, if a new message is defined that goes from the AE unit named "A" to "B", then "A" and "B" consult their connection table looking for an existing connection using the same protocol. The supported protocols are TCP and UDP. If one exists, then no action is required. If, on the other hand, a connection does not exist, then a new one is created and information about the AE name, IP address and the network channel number is inserted in both parties' tables.

The circular pipeline message passing construct (Figure 5) was added late in the development process of the AE system. The existing networking code for the project

contained a few weaknesses and a bug surfaced when the circular pipeline construct was added. To explain the problems requires a basic knowledge of how a TCP connection is created. This will be outlined below.

TCP is a connection-oriented protocol. For two AE units to establish a TCP connection, one side (the server side) must create a socket and then "listen" on that socket for connections. Meanwhile the other AE unit (the client side) must also create a socket and then through that socket it attempts to connect to the server side (using IP address and port number). Timing is a critical aspect in the above sequence of events. For example, if the client attempts a connection before the server is ready and listening. then the client's connection attempt will fail. On the server side, the listener will wait indefinitely for a connection unless special socket options are used to cause a listener to time out.

The earlier code for the AE project attempted to deal with the problems listed above by using less than ideal solutions. The old technique used, described below, gave the server side of a network connection a small time advantage over the client side. The time advantage was provided by the UI's action of sending server side connection requests to the AE units before the corresponding

client side connection requests. For each arc in a message's voyage (see Figure 5), the UI would send two connection requests commands, one for the receive side (server side) and one for the sending side (client side). For example, a circular pipeline message with three AE units (i.e., A -> B -> C and back to A) would produce six connection request messages from the UI: three send and three receive. Each AE unit for the circular pipeline case just described would receive two connection requests: a send and a receive request. This usually worked by allowing the server side "some" extra time to establish its socket before the client side attempted to complete the connection. The advantage of starting earlier usually solved the timing problem, but because it did not eliminate the timing issue, the code occasionally failed.

The circular pipeline was added, because, without it the AE system did not easily support two way communications. The most common form of communication is two applications communicating. For example, "A" sends a message to "B", "B" processes the data and sends a response back to "A". The circular pipeline made this (and more complicated communication topologies where the originator receives a response back) much easier to construct.

The addition of the circular pipeline message-passing construct caused a deadlock when using the older network code. Each AE unit processes network connection commands serially, and because the UI made sure server side connections were processed first, all AE units involved in a circular pipeline message were acting as servers waiting for a client connection. However, the same set of AE units waiting for a client side connection were the ones that needed to also act as clients. The result was a deadlock situation.

Consider the following example. If "A" and "B" are involved in a circular pipeline connection they will both receive two connection requests, a server side with the other AE unit and a client side with the other AE unit. They first execute a blocking call to listen for a connection (server side) and then wait. If the *listen* finished, they would next execute the client side of the connection but because neither is acting as a client and the server side will wait indefinitely. The result is a deadlock.

After this problem was discovered, the entire networking code was reevaluated. The changes included: multiple retries on client side connections (including progressively longer times between retries), server side

timeouts (if a client never connects, the server side will give up) and multiple threads to process connection requests. The use of multiple threads allows an AE unit to service client and server side connection requests simultaneously. This fixed the deadlock situation. These changes fixed all the known problems with the networking code.

i) Network, Send and Receive

This section describes the three modules that allow network communication between AE units. The three modules are grouped together because of their interactions and common functionality, but they are distinct software modules. The network module was written in Ada95 (as was the rest of the AE system) but the send and receive modules were written in the C programming language because of its flexibility and system interfaces.

The network module controls the networking functionality for the AE. Its main functions include: processing of new connection requests, checking for new messages, preparing messages for transmission, and recording

data metrics on messages. Each of these areas is discussed below.

The processing of new connection request allows the transmission of messages between AE units. When an AE unit is initialized, it first creates a TCP/IP connection with the UI. The UI reads the command file, and when it processes network commands, it sends connection requests to the appropriate AE units. The networking module acts on these requests and creates the necessary networking connections between other AE units. When an AE unit has active network connections, it periodically polls those connections to check for the arrival of messages. The networking module maintains a list of active connections and periodically calls the receive module (described below) to check for messages. If a message is received, the network module performs the following actions:

- insert a time stamp into the message (time received)
- increment a counter recording the number of messages received
- add the byte count of the current message to the total byte count for the protocol (i.e., TCP)

The send module receives its input from the network module (described above). Its job is to package the three components of a message into a buffer, and pass the

message to the Operating System (OS) for transmission. Figure 7 diagrams the three parts of a message. The amount of padding is the total size of the message minus the other two parts: header and AE network command data structure (the data structure is diagramed in Figure 6).

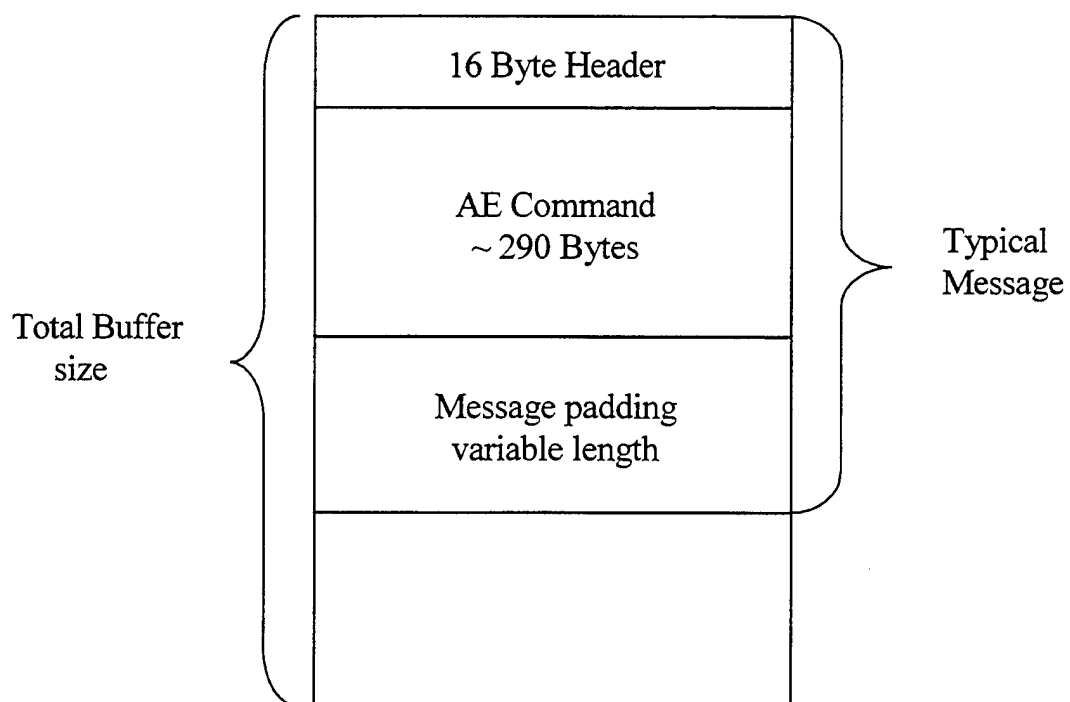


Figure 7 Message Layout

One of the problems associated with a wide range of message sizes is maintaining buffers. Message lengths

are dynamic, and it is possible that the next message will be larger than the current buffer. The send routine maintains separate buffers for sending UDP and TCP messages. The first step in building a message for transmission is testing the message buffer's size against the input parameter that defines the current message's length. If the current message buffer is not large enough to hold the current message, then a new buffer is allocated and the existing one is released. To minimize memory allocation/de-allocation, (generally considered a problem due to memory fragmentation recovery processes that can cause real-time systems to miss deadlines in a real-time) system, the following technique is employed. The new buffer is five kilobytes larger than the current message. The value of five kilobytes was arbitrarily chosen. Although the increase in size is much larger than needed for the current message, the overhead and impact of memory allocation is minimized.

A message, as diagramed in Figure 7, is constructed from the top down. First the header, a sixteen-byte field is built and copied into the buffer. The message header is described in detail below in the receive module section. Next, the instructional part of the message is copied into the buffer (this is the information that comes

from the AE command via the command file). The message padding is not formally placed into the buffer. Because the system call to send a message requires a pointer to a buffer and the message's length in bytes, the padding is safely included in the message by ensuring the buffer is larger than the message size.

The last module covered in this section is the receive module. It is the receive module's job to undo what the send module built up and then to return the AE command data structure to the message processing module (covered in section III.C.3.j).

There are differences between the communication protocols TCP and UDP that require the receive module to treat these protocols separately. TCP messages are received as part of a flow of information that spans messages. UDP messages, on the other hand are received individually with no overarching organization imposed upon a series of messages. In between the sender and the receiver, the network components may break up a UDP packet into separate IP packets but the receiving side's OS will deliver the same size message to the receiver.

To receive UDP messages, the receive module calls the *recvfrom* system call. One of the parameters to the *recvfrom* system call is the number of bytes to read. If

that number of bytes is less than the entire message, then part of the message will be lost. For example, if we receive a 500-byte UDP message and only read the first 100 bytes, then the last 400 bytes are lost and cannot be read later. This feature actually helps the AE receive messages because the number of bytes that must be read is known (header and AE command data structure) and the remaining bytes can be safely dropped.

When receiving TCP messages, the receive module needs to maintain message boundaries. Here, the main problem is that a receiver does not know the length of a message before receiving it, and, unlike UDP, all the bytes of a message must be read before the module can process future messages. The header, introduced above solves the problem by providing the message size to the receive module. A message header contains the following information:

- message size in bytes,
- message type and
- endian field (described below).

The receive module will first issue a read to obtain the header information, it can then calculate how many additional bytes of information must be read to fully receive that message. Next, it will read the instructional

part of the message into a data structure that will be returned to the calling procedure. The final step is to read the remaining bytes of the message. These bytes are formally known as the message padding, and they are read and discarded.

The network receive module uses a unique and fair method for processing messages over multiple active network channels. The fairest way to process messages would be as they were received. Unfortunately most operating systems do not instruct an application that has more than one pending message any timing information on those messages. Fair means that if the last message received was from Connection Channel Three, and now the AE unit has two messages ready for processing (one on Channel Three and one on Channel Five), then we will process the message from Channel Five. The implementation uses an integer to remember the last active channel. When more than one channel has a message ready for processing, the AE uses a modular counter to select the next message for processing. that is, the AE unit will choose the channel numerically higher than the last one selected (the selection will wrap around to zero if the last one selected is numerically the highest in the set). It is the author's observation that most communication software, using the *select* system call will

favor lower number (over higher) channels when two or more messages are ready for processing simultaneously. The technique developed for the AE appears to be unique.

To summarize, the network modules take care of many issues related with communication over networks and allow the AE system to emulate complex message passing applications. The sending modules build up the messages for transmission. The receiving module processes the header information to 1) deal with 'endian'³ issues, 2) identify the message and, 3) by using the size information, safely receive any size message. After a message is received, it is returned to *Message Processing* for further processing.

³ Endian refers to one of the many data compatibility issues that can occur when computer systems from different manufacturers or operating systems communicate over a network. The endian problem stems from the fact that some data types are stored differently on different computers. Big endian systems store the most significant part of the number of some data types first (lower address value) and little endian systems store the values in a reversed manner [STEV98]. The endian field (borrowed from HiPer-D) provides a nice method for a receiver to quickly determine if the message received needs an endian conversion. The field contains a value that when tested informs the receiver if the message requires an endian conversion or is fine as received.

j) Message Processing

The input to this module is the output from Network receive component (section III.C.3.i). The receive module returns known data types (i.e., AE network commands) and places them on a circular queue. The message-processing thread is event-driven and if no messages are available for processing it stays in a blocked state (to reduce CPU usage). If the queue is empty then the message-processing module remains blocked. The event of adding an item to the queue unblocks the message processing thread. This feature is implemented using ADA95 protected objects. Protected objects operate provide mutual exclusion. The rest of this section contains a description of how messages are processed by AE units.

The design decision to separate message reception from message processing allows the receiving thread to efficiently receive pending messages. The processing of messages can be time consuming, and is therefore handled by a separate thread. The following steps outline what each AE unit does to process a message:

- compute and complete CPU workload,

- forward the message if necessary, and
- execute an event if necessary.

Each message contains workload information for each receiver (workloads are uniquely defined for each receiver of a message). The pipeline, circular pipeline and fan-out (Figure 5) message constructs are examples of messages that can have several receivers. Workload information is defined with the same parameters as the CPU loader, and therefore it is described as a statistical distribution. The workload emulation uses the same benchmark module as the CPU loader module. The workload is used to simulate the work involved in message reception and processing. Next, the message is checked to determine if it should be forwarded. A pipeline message (Figure 5) is an example of a message that some AE units (B and C) would need to process and then forward. If this AE unit is the last receiver of a message, then an optional event can be included. Events can be:

- start a CPU loader job,
- stop a CPU loader job, or
- send a new message.

All events have an associated probability. This gives the AE system the ability to dynamically alter its own behavior and fulfills the requirement of supporting transient periodic processes.

k) Controller

The Controller provides the interfaces between the UI and the internal modules of an AE unit. Its major function is to receive commands from the UI, process those commands and then issue the commands to an appropriate module within the AE unit. Further, the controller reports new information to the UI. There are four types of commands that the controller has to process: CPU, memory, message, and shutdown. Once a command is identified (e.g., a CPU command) it is sent to the appropriate module for processing. A CPU loader command, for example, will create a new CPU loader process. Because all timing issues related to commands are handled by the UI, the controller merely processes commands when they are received.

D. SUMMARY

This chapter has described the AE system. It started with a high level view of the AE system as it would be used as an emulation tool (Figure 1 AE System). Next that system was examined at a component (i.e., AE unit, see Figure 2) and at a sub-component level. At the sub-component level, many of the details about the AE unit were explained. In addition, some of the problems encountered while developing the AE system were also discussed.

The next chapter will present results from a series of emulation experiments using the AE system. A tactical modeling tool was used as the target application for the emulation. The results show that software emulation using the AE system can be effective.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SYSTEM EMULATION AND EXPERIMENTATION RESULTS

A. INTRODUCTION

This chapter describes how the AE system can be used to emulate an existing software system. The emulation process has three major steps. For demonstration purposes, an example that emulates a system from Teledyne Brown called EADSIM [EAD00], is used. Before the steps are described, an overview of EADSIM is presented. The final section of this chapter describes the work required to validate the AE system's accuracy in emulating a real system.

B. EADSIM

Extend Air Defense Simulation (EADSIM) is a warfare modeling program. EADSIM is widely used to model battle scenarios as an aid in making tactical decisions. It consists of four modules: C3I, Detection, Propagation and Flight Processing. These modules operate in a distributed fashion and thus use networking protocols to communicate.

One of the four modules is optional (i.e., Propagation) and was not included in Porter's [PORT99]

thesis results, which are used as input to the AE's emulation process. EADSIM supports a wide range of tactical systems that can be included in a model. Battle scenarios are constructed through a complex iterative process [PORT99].

For the purpose of this thesis, the configuration of, and results from, EADSIM are interesting but not necessary. Remember that the AE system does not return useful results, but rather loads the system as if a useful application was running. The block diagram of EADSIM (Figure 8) shows the communication paths between the three distributed modules of EADSIM. The resource usage results from Porter's execution of EADSIM are presented at the end of the chapter, followed by the results of the AE systems' emulation experiment using EADSIM.

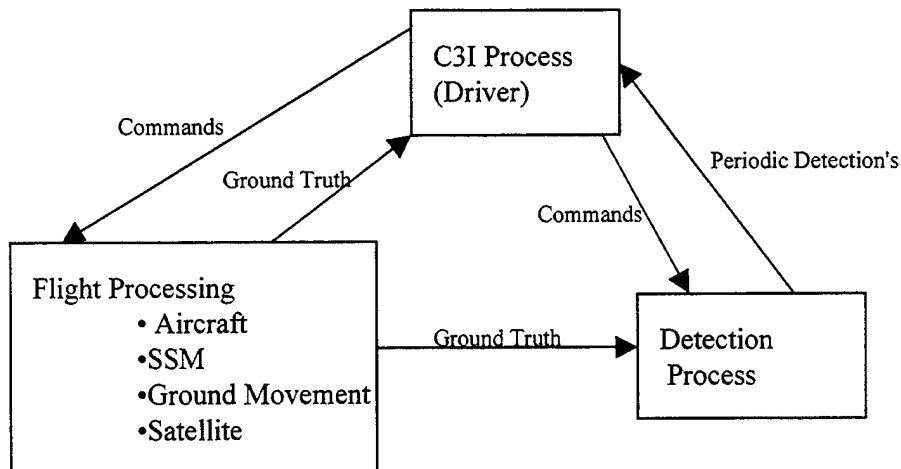


Figure 8 EADSIM Runtime Block Diagram

C. SYSTEM EMULATION: THE THREE STEP PROCESS

Starting with a system like EADSIM, and automating the steps to emulate it using the AE has always been a desired feature of the AE project. Figure 9 below, shows the three-step process for creating an emulation from an existing system using the AE system. For reasons listed below, the goal of automating this process was not realized. A problem was the tools (or lack of tools) needed to profile an application's components to produce the information required to construct the AE command file. Also, as will be shown, a

general understanding of the system under study is required and cannot be obtained from the profiling tools.

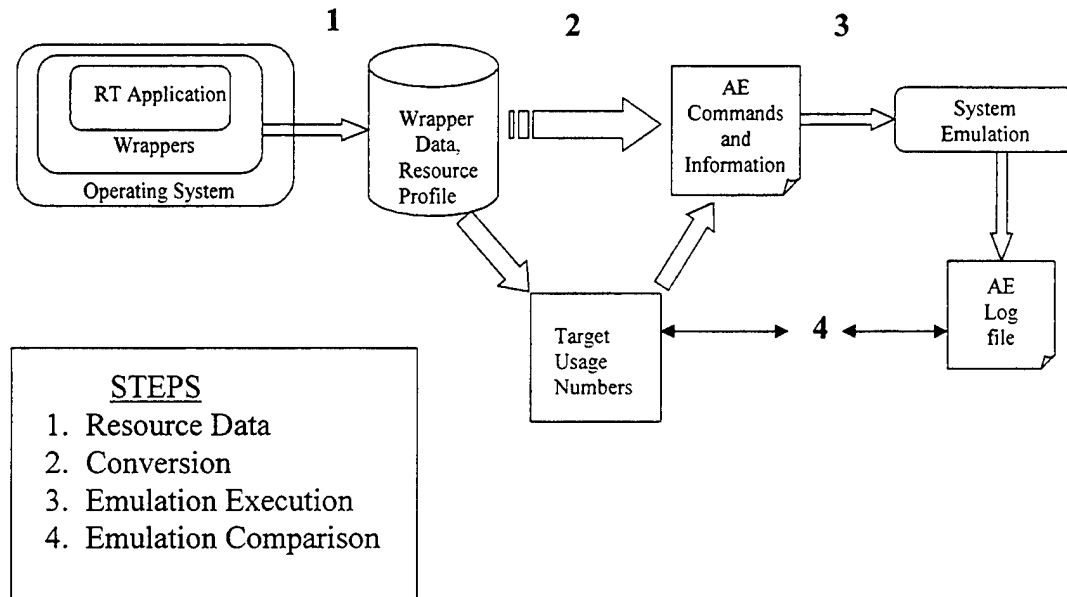


Figure 9 Emulation Steps

Figure 9 diagrams the steps involved to use the AE system for emulating an existing system. The fourth step, not addressed in this paper is a response loop, which allows the emulation process to be tuned. The steps shown in Figure 9 will be described for emulating the EADSIM application by the AE system.

1. Step One: Gather Resource Usage Data

The objective of this step is to gather data on the use of system resources by the application. As shown in Figure 9 the wrapper tool was used for component profiling; this profiling tool was developed by Schnaidt [SCHN98] under the MSHN project (MSHN wrappers). The MSHN wrapper tool operates between an application and the Operating System (OS), by intercepting system calls. Here a MSHN wrapper is a low-overhead component that usually only records the parameters to a system call. For example, a network *send* calls the OS *write* function, the MSHN wrapper interrupts the *write* function and records the number of bytes followed by a call to the underlying OS *write* function. All applications (on Unix) call the *exit* function to halt normally. As implemented by Schnaidt, the MSHN wrapper for the *exit* function completes its job by obtaining the CPU usage data and logging all the resource usage data collected.

The MSHN wrappers provide network and CPU usage profile data. All the data provided in this chapter on EADSIM was compiled by N. Wayne Porter and was obtained from his thesis [PORT99]. These data are then used as input into Step Two, outlined below.

2. Step Two: Using Profile Data to Construct AE Commands

Starting with the profile data from the previous step and creating AE commands that accurately emulate a system is the most difficult step in emulating an existing software system. Some of the data provided by the MSHN wrappers can be easily converted into AE commands, while other data require a conversion step. This section will cover the details of converting the MSHN wrapper data into AE commands.

The network data are in a format that maps nicely into AE commands. The MSHN wrappers report networking data in the following areas:

- total number of messages sent,
- total number of messages received,
- total bytes sent, and
- total bytes received.

The conversion from the above format into AE commands is fairly easy because both systems use similar units. The AE system sends messages through a CPU loader module's action. Thus, if an application sends 4 messages per second and the CPU loader has a period of 0.5 seconds, then in each

period, the application will send two messages. If the number of messages varies, then the probability (of executing the action) and the repeat value (for the CPU Loader) can be modified to obtain the desired message rate. The AE message command provides the AE unit with the following information: size in bytes, and path information. The transmission rate of a message is related to a CPU loading parameter. See Section III.C.2 for a more complete description.

The CPU workload information currently provided by the MSHN wrappers does not easily convert into a format that can be used to construct AE commands. The MSHN wrappers report CPU utilization in seconds (e.g., 17.115 CPU seconds) for each module. An AE unit, on the other hand, operates in terms of Kilo-Whetstones (1000 Whetstone instructions) or Kilo-Dhrystones. The mapping between these units, to any degree of accuracy, requires executing an AE unit on the same computer used to obtain the MSHN wrapper data.

The method developed to address this uses the AE system's percentage capability to output the number of Kilo-Whetstones needed to utilize the CPU at 100% for a specified time. To specify the CPU usage in that manner requires a single command to run a CPU loader module for one iteration using 100% of the CPU for the amount of time desired (e.g.,

usage = 100%, number of iterations = 1, time period = 22.567 seconds). The AE unit will then print out its calculation for the number of Kilo-Whetstones or Kilo-Dhrystones needed to produce the desired result. Remember that these are ideal results. If you actually programmed an AE unit to execute with those parameters it would execute that many Kilo-Whetstones, but it would most likely not finish in the time specified.

The MSHN wrapper provides the number of CPU seconds the application used and this figure is used as the number of seconds to use 100% of the CPU in order to emulate the application's CPU usage. The number of Kilo-Whetstones returned by this method becomes the target number of Kilo-Whetstones for that module to execute over the entire emulation.

The algorithm used by an AE unit to calculate the number of Kilo-Whetstones needed to consume a percentage of a CPU is described below. When an AE unit is asked to use a percentage of the CPU (e.g., 35%) it first does a test designed to use 100% of the CPU for a short period of time which produces a usage value that is used for all percentage calculations. The test must take into account the following assumptions and problems associated with clock granularity.

- For short time periods one user gets 100% of a CPU.

- Tests that are short have problems because clock granularity can introduce error.
- Testing several times increases the odds that all tests will not be pre-empted and swapped.
- Longer tests reduce the clock granularity problem but increase the preemption problem.

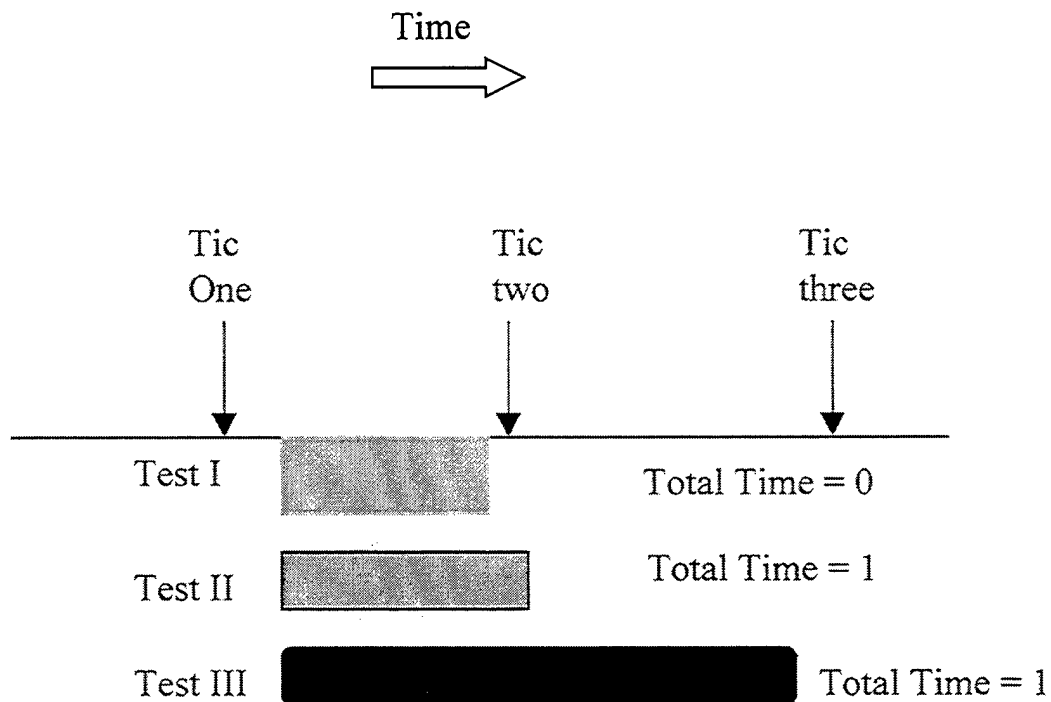


Figure 10 Time Granularity Example

Figure 10 shows how clock granularity can introduce problems (or error) into calculations. The basic problem

stems from the way the system reports time. In the area between *Tic One* and *Tic Two*, the system will report the same value for the current time. Timed events that operate for short periods, relative to the clock granularity, can lead to misleading results. Test I and II are almost the same duration but return values that would create different assumptions about their performance. Test III is almost two clock tics in length but is reported as being one clock tick. In this example test II is the only one where the reported time is close to the actual time.

For a given CPU (i.e., computer), we wish to calculate the number of KW (Kilo-Whetstones) that can be executed in one second. Two of the values, number of KW (20,000), and number of times to execute the test (i.e., 7) were selected while accounting for the problems listed above. The test recorded the start time, *ts*, and the time at the end, *te*.

$$\frac{xKW}{1\text{sec}} = \frac{MKW}{(\text{test})\text{sec}} \Rightarrow xKW = \frac{MKW}{(te - ts)} * 1\text{sec},$$

Where

xKW: total Kilo-Whetstones needed to use 100% of
the CPU for 1 second
ts-te: elapsed time
MKW: number of KW used for the test (20,000)

The timing performance built into the AE system operates at the millisecond (ms) time interval. As stated

above, an AE unit can be programmed to consume a percentage of the CPU (i.e., 1-100%). Therefore, a CPU loading task can be programmed to operate with a period of x ms that will consume y percentage of the CPU. The formula below calculates the number of Kilo-Whetstones that will consume the desired percentage of the CPU, for the time interval specified. The term xKW from the previous formula provides the baseline for this calculation.

$$yKW = T * P * xKW ,$$

Where

yKW: CPU workload in KW

T: time in ms (ex. 10ms is entered as .010)

P: percentage (25% is entered as 0.25)

xKW: The value of KW that will use 100% of the CPU for one second

A short example will illustrate the calculation. If xKW is 100 and a user wants a periodic CPU load that uses 50% of the CPU and operates with period of half a second. It is easy to see that the answer should be 25. The formula becomes: 0.5 (time) * 0.5 (percent) * 100 (xKW) and will yield the correct answer.

The percentage usage option for the AE was shown to operate as designed. The Unix top command was used to verify the percentage usage, because it reports an application's CPU usage as a percentage. For the

experiment, the CPU load was set to 40%, and a time period was set to 1 second. The *top* command, reported the AE CPU usage with in 1% of the desired value. The results validate that the method and parameters selected produce the desired CPU loading.

As was shown, the conversion from CPU seconds to Kilo-Whetstones is possible by programming an AE unit to use 100% CPU utilization for the length of time reported by the MSHN wrappers for CPU usage. The data for the conversion of EADSIM modules from CPU seconds to Kilo-Whetstones is contained in Table 1.

When an application's CPU workload is expressed in Kilo-Whetstones, it can be converted into the command language that drives the AE system. It should be noted that this method is simplified from the normal case. Most applications will have several threads, and detailed information about each one may be necessary to fully emulate the application. The data obtained from the MSHN wrappers does not give any details about how many threads were operating and the CPU usage of each thread.

Although the CPU percentage usage is part of the CPU emulation capability of the AE system, it was not originally included for the following reason. Applications can be profiled by percentage CPU usage but what they actually do

is complete a task or set of tasks. If an application is ported to a different computer system then that same application may finish in a shorter time while using a smaller percentage of the total CPU capacity. The AE system's CPU emulation is centered on the idea that applications complete tasks and using a percentage of the CPU does not allow the AE to illustrate performance variations in different computers and operating systems. The AE uses a synthetic workload (Kilo-Whetstones) to represent (or emulate) actual workload.

Another challenge to the conversion of resource data into parameters for the AE command language was that the MSHN wrappers do not record any real-time information. Real-time information must be obtained through an understanding of the system under study.

In conclusion, by using the information provided by the MSHN wrappers, and a working knowledge of the system under study, it is possible, by using various conversions, to build the commands for the emulation (i.e., an AE command file).

3. Step Three: Running a System Emulation

This step involves taking the command file produced in the previous step and executing it to emulate the original system's resource usage profile. The first step is the process of starting and synchronizing all the AE units. When all the components (the UI and all the AE units) are operational, the UI begins reading and processing the command file.

This section contains some detailed information about the startup process that was introduced in Chapter 3. Some of the details in this section review that material.

Each AE unit supports several command line parameters, but only two of them play a role in the distributed architecture (the others allow an AE unit to operate as a standalone CPU loader). The first parameter defines the AE unit's name, and the second parameter contains the hostname of the system where the UI is operating. The UI also has two command line parameters of interest: *command-file*, which contains the commands used to configure the AE units; and an *integer parameter* which informs the UI as to how many AE units are participating in the experiment. The command file

contains all the commands that give each AE unit its identity (to emulate its part of a RTDS). The parameter informing the UI of the number of AE units allows the startup process to take an indeterminate amount of time to complete. The UI keeps a running count of AE units and waits until they all have an active connection with the UI before starting an emulation experiment.

Figure 11 diagrams the automation for running experiments. There are two levels of processing above the AE system level. The top level written for this thesis starts the automated startup level and, after AE system completes, this script will copy the remote data files into a file structure defined in the script's configuration file. The automated startup script (see Figure 11), borrowed from the DynBench project, starts the components (the AE units and the UI) and supplies them with their command line parameters. Using remote authentication⁴ for starting processes, the Startup script can start processes on any computer system on the LAN. The configuration files for the two tools are included in Appendix C. Figure 11 contains a graphical representation of the tools.

⁴ Remote authentication allows user and system pairs to be mutually trusted, and, as such, can execute commands without presenting a password as might be required in an interactive session. [UNIX97].

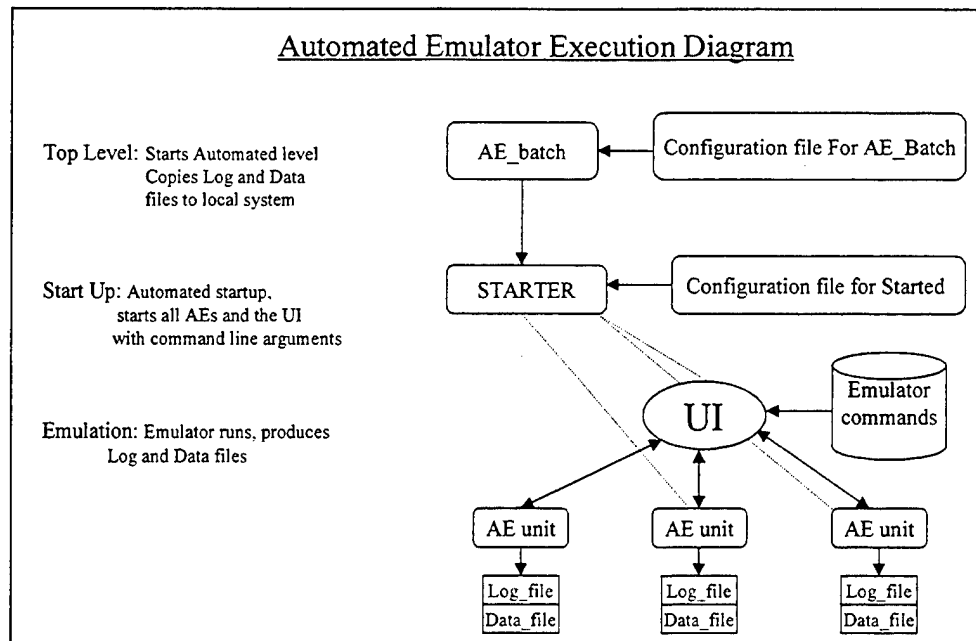


Figure 11 Automated Emuation Diagram

Once all the necessary AE units for an experiment have connected with the UI, the emulation process begins. All AE commands have an optional time parameter, which is based on the time that the last AE unit established a connection with the UI (i.e., elapsed time). As described earlier, the UI processes the command file, and then issues each command to the appropriate AE unit. Normally the last command in the command file, is the *stop_all* command, which instructs all the AE units to perform a normal shutdown. Before shutting down, each AE unit outputs all its data and debugging

information to a data and log file. The data file contains the following data relevant to QoS considerations:

- network usage data
 - number of messages sent and received
 - total number of bytes sent and received
 - timing information on each message
- CPU usage data
 - total number of Kilo-Dhrystones executed
 - total number of Kilo-Whetstones executed
- deadline information
 - Each CPU loader module records the number of deadlines missed

The next section compares the information obtained from experimental runs of EADSIM (the data from the AE system is obtained from the data files) with the target numbers for the emulation of EADSIM.

D. EADSIM WRAPPER RESULTS

The data in Table 1 was obtained from Porter's [PORT99] thesis. He obtained the data from the MSHN wrappers while executing EADSIM as shown in Figure 8.

Table 1 EADSIM Resource Usage Data

EADSIM Resource Usage Data			
	<u>C3I</u>	<u>FP User</u>	<u>Detect</u>
User CPU time	17.717	17.125	16.316
System CPU time	3.026	3.196	5.855
Total CPU time	20.743	20.321	22.171
Wall time	94.5	77.1	93.3
Bytes written	1,634,436	1,029,378	2,057,529
Number of writes	155,957	741	589

The three data columns in Table 1 are labeled by EADSIM modules (see Figure 8). As shown in Table 1 the wall time (i.e., actual execution time) is much longer than the CPU usage time. It is important to note that EADSIM is not a real-time application, but is similar to a real-time application in that its operations are time stepped [PORT99]. The importance of time makes sense because a battle simulator must account for when and where events happen. Table 2, contains the conversion from the MSHN wrapper CPU data into Kilo-Whetstones. The Kilo-Whetstones numbers are the target CPU usage numbers for the three components in the emulation.

Table 2 CPU Resource Usage Data for EADSIM

EADSIM CPU Resource Data and AE Conversion CPU Data			
	<u>C3I</u>	<u>FP</u>	<u>Detect</u>
CPU time	20.743	20.321	22.171
Kilo-Whetstones	906,096	887,662	968,474

Table 3, contains the network information from EADSIM. Included is a new row that shows the average message size transmitted by each component.

Table 3 Network Usage data from EADSIM

EASDIM Network Data			
	<u>C3I</u>	<u>FP</u>	<u>Detect</u>
Number of writes	155,957	741	589
Total bytes	1,634,463	1,029,378	2,057,529
Ave. msg. size	10.5	1389.2	3493.3

As shown in Table 3, the average message size sent from C3I was between 10 and 11 bytes. The AE system's minimum message size is approximately 300 bytes, due to the overhead introduced by the complexity of the AE messaging. The result is the AE cannot emulate small messages. A compromise to permit emulation of C3I's network traffic was to lower the total number of messages while increasing the

size of the average message to a value that the AE system could support. The resulting emulation reasonably matches the number of bytes sent by C3I, but not the number of messages sent.

EADSIM's operation is time stepped. C3I controls the execution by issuing commands, which include timing information to the other modules (i.e., Detection and FP). When the other modules complete the workload for current time step they send information back to C3I, and the process repeats until complete.

Using the AE system to accurately emulate EADSIM will require the same master/slave relationship. Because the execution of EADSIM is time stepped, the two processes that are slaves (Detection and FP) to the master (C3I) will receive all their workload via AE messages. The method of emulation was intended to simulate actual operation, where the three components of EADSIM complete one time step's work and then wait for the command to start the next time step.

The algorithm used to construct the command file for emulating EADSIM is described below. The command file used for the emulation is included in Appendix B. The wrappers provided high-level usage information about the three modules of EADSIM. The wrappers did not provide detailed information about EADSIM execution characteristics. For

example, it was not possible to tell whether or not the CPU usage of FP occurred evenly over the execution time or if it had periods of greater usage and other times of much lower than average usage. Without detailed usage information, the emulation was forced to assume that the CPU usage was consistent over the wall clock time of the execution of EADSIM (94.5 seconds). EADSIM is driven by the C3I process; it provides the timing, through commands to FP and Detect. Therefore, the emulation will focus on C3I process and use the same architecture to drive the other two modules. The easiest way to construct the emulation was to treat the relationship between C3I and detect separately from the relationship between C3I and FP. Therefore, the CPU load for C3I will be divided in to two CPU Loader tasks. One CPU Loader task sent messages to FP and the other sent messages to Detect. Both messages contained the workload information necessary to emulate CPU usage for FP and Detect. Other messages that are part of the EASDIM system were emulated using events. Table 4, lists emulation target values and experimental results. It is important to remember that the target numbers are emulation target numbers and are not MSHN wrapper results. The target numbers were established through the conversion process described earlier. Recall also that C3I's number of

messages sent was modified to account for a shortcoming in the AE system.

Table 4 Target and Emulation Results

Target Numbers for EADSIM Emulation			
	<u>C3I</u>	<u>FP User</u>	<u>Detect</u>
Kilo-Whetstones	906,096	887,662	968,474
Messages sent	5,448	741	589
Messages received	1,330	2,724	2,724
Bytes sent	1,634,463	1,096,533	2,057,529
Bytes received	3,086,907	817,232	817,232
Experimental Results From AE System Emulation (Averages)			
Kilo-Whetstones	925,979	887,365	967,881
Messages sent	5,441	781	566
Messages received	1,345	2,720	2,720
Bytes sent	1,719,738	1,096,533	1,984,406
Bytes received	3,059,339	816,082	816,586
Percentage Error (absolute value)			
Kilo-Whetstones	2.19%	0.03%	0.06%
Messages sent	0.13%	5.40%	3.90%
Messages received	1.13%	0.15%	0.15%
Bytes sent	5.22%	0.00%	3.55%
Bytes received	0.89%	0.14%	0.08%

Figure 12, shows how the three emulation steps, and tables, presented in this section fit together. The data in Figure 12 is the average values for the resource loading. The exception is the *target numbers*, which are calculated from the resource usage data. The emulation experiment shows that the AE system can be used to emulate existing systems, and that it can produce results that are within a

small percentage of the target values. It is important to note that all emulations will have some variation from the actual resource loading of the system being emulated. For this experiment, the amount of variation from the target numbers as shown in Table 4 was small.

The results presented are from a sample size of 103 emulation runs of EADSIM. Appendix E shows a full spreadsheet containing the data collected from the 103 experimental executions of the AE system emulating EADSIM. As can be seen by examining the data in Appendix E, the results for most of the metrics are fairly close to the average for all runs. The data in Table 4 contains the average values for all the metrics recorded.

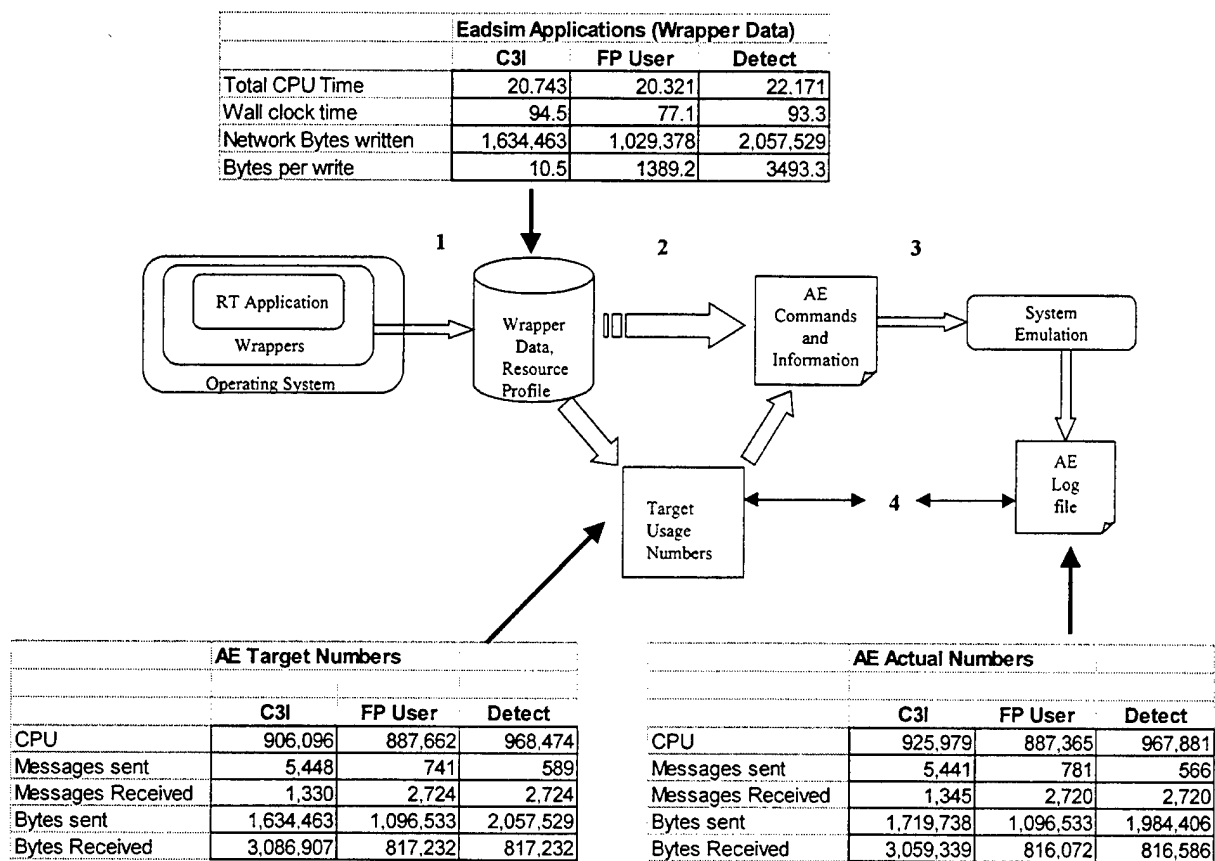


Figure 12 Experimental Results Diagram (Averages)

Additionally, the results show that the AE system operates as intended. The commands were carefully written for this experiment, but if the AE system had not operated as intended then the results would have showed a larger percentage error for one or more of the recorded metrics.

E. SUMMARY

This chapter showed how an existing system can be profiled by starting with data obtained using the MSHN wrappers. Further, we described how that MSHN resource usage data can be used as input into a process that can build all the necessary configuration files for an emulation using the AE system. As was shown, the emulation can then be executed and the results obtained from the AE's data files can be compared to calculated resource usage values. The results obtained showed that the AE system did accurately emulate EADSIM resource usage. Adjustments can be made to compensate for the AE limitations, for example the fact that the smallest size of an AE message was much larger than that of the application it was emulating.

THIS PAGE INTENTIONALLY LEFT BLANK

V. DISCUSSION AND CONCLUSIONS

A. LESSONS LEARNED

Some potential customers of the AE were not comfortable with the Ada programming language selection. Ada is perceived to be a government mistake and therefore most sites do not have the expertise or compilers to support Ada development. The choice to develop the AE in Ada95 was a good technical decision but possibly a poor one for marketing the AE.

Much of the AE was developed using Object-Based programming and not full Object Oriented (OO) techniques. "Object-Based usually refers to objects without inheritance and hence without polymorphism" [OBJFAQ]. If the AE project was designed and developed using full OO features then future changes could easily produce new and powerful capabilities, while leaving the existing functionality intact. The current design allows for change but does not leave the old functionality intact. A full OO implementation would have been a wise decision.

B. FUTURE WORK

A number of additions to the AE would increase its emulation functionality. One improvement would be to provide a general mechanism to allow the AE to send and receive messages from existing systems. Used in this manner, the AE system could obtain its loading from an existing system or be used to drive an existing system. While this capability exists, it is limited and must currently be customized for each type of message. A general-purpose method for this type of feature would be of great value for a real-time development project.

The use of multicast could reduce the complexity of the UI to AE unit communication. Using multicast for the UI-to-AE communication would eliminate the command line parameter to the AE used for finding the UI. This feature would also help the implementation of migration of AE units while an AE system is operating (one of the features not yet implemented).

Other future features include different load simulators for many of the other resources that applications utilize. The following list contains some of the resources that would increase the emulation capability of the AE.

- File access (local and file server)
- Display subsystem
- Database

C. COMPARISON WITH RELATED WORK

This section contains the comparison of this thesis and other projects that are closely related with the AE system.

1. DynBench

A common goal for the AE and DynBench projects was to provide researchers tools with which to emulate the HiPer-D system. The approaches taken by the two efforts were vastly different. DynBench's approach was to build a simplified version of HiPer-D, making it a specialized solution to the problem. The AE system on the other hand, is a general-purpose real-time application emulator, and because HiPer-D

is in the class of systems that the AE can emulate, it too can provide an emulation solution. The primary task in creating such an emulation would be the construction of the command and configuration files.

The HiPer-D team plans to combine the DynBench and the AE system and make the combined system available to other researchers. Users will be able to use the AE and DynBench either in combination or individually. These two systems are complementary. The AE system offers users a wide range of configuration options while DynBench offers users a specific and well-tuned HiPer-D emulation tool.

2. Carff Emulator

Carff's emulator has many interesting characteristics. It is a distributed, portable (developed in Java), message passing application emulator. It contains many of the high level features found in the AE system, but its code size is at least an order of magnitude smaller than that of the AE.

There are several differences between the two systems. The main one is that the AE is a real-time emulator and Carff's is a user-level application emulator (applications that execute a task and finish). The message passing subsystems are vastly different; the AE supports a complex

yet flexible message passing subsystem, while Carff's only supports point-to-point messaging. The CPU workload of the two systems is similar; both offer a wide range of options for providing CPU loading with statistical variation.

By using the programming language Java that abstracted out the details of networking, the Carff emulator enjoyed a much shorter development cycle than the AE. In contrast, networking is at the heart of the AE project. For the AE, networking took the lion's share of the development time and introduced most of the difficult problems.

3. Petri Nets

Petri Nets are a tool that allows researchers an indirect method for studying systems. The method includes building a mathematical model of the system under study. This model is then studied in a laboratory setting. This indirect method of study is useful when the actual system is difficult to study.

The AE will allow modeling through emulation, and, as such, will allow Petri net-type analysis of some systems. Using a loose definition, the HiPer-D system is an example of a Petri net system. In this case, it is safer and easier to develop and study the system in a lab before fielding it

on a ship, where lives and operations will depend on its functionality.

4. Hartstone

The Hartstone benchmark [HART89] is a tool that can be used to prototype real-time systems and is mainly used for studying real-time system performance. There are many similarities and differences between the use of the Hartstone benchmark system and the AE project.

Starting with the similarities, both systems can support:

- Prototyping of real-time systems,
- Sending and receiving of messages,
- Periodic and aperiodic tasks and
- Synthetic workloads.

The differences between the two tools are numerous. The Hartstone benchmark is intended to operate as a single system that will return a performance metric. The metric is either on (the system has met all its real-time deadlines) or off (the system missed at least one real-time deadline). The AE, on the other hand, was intended to be a tool that operates concurrently with other systems. Its main purpose

is to allow experiments to determine the effects of CPU loading and network communication on the total system. Many of the other differences stem from that difference. For example, a Hartstone test will terminate when a deadline is missed. The AE simple records the event and keeps on executing. The messaging subsystem in the AE reflects the recent growth in distributed systems where communication is not always point-to-point. It allows for messages that span several applications and further records the time it takes that message to traverse its path. Another big difference between the methods relates to workload, the Hartstone benchmark defines workload in terms of percentages while the AE uses an actual value (i.e., kilo whetstones) as well as percentages.

The AE implements or is designed to support many of the latest developments in real-time software. For example, application migration would not be supported by the Hartstone benchmark. The Hartstone Benchmark is primarily for embedded real-time systems [HART90]. It would be almost impossible to meet a deadline if an application were to migrate during a period. The new approach is to allow, a system experiencing problems to miss some deadlines while a controlling application (i.e., a RMS) carries out an effort

to return the system to full functionality as quickly as possible.

It would not be difficult to convince someone that a system that is critical for an airliner's operation should be able to recover from an event such as a PC crash. This is an example of a "real-time mission-critical system that must respond in a timely manner to conditions in their environment" [WELC98]. The recovery process might merely require that applications that existed on the crashed system be moved to a different computer. Thus, the whole system could be restored to full operational status. In this scenario, the crash may cause some short term problems, but if the remedy is applied before total control of the aircraft is lost, then the safe recovery can be achieved. The AE is a tool that can support this paradigm and the Hartstone benchmark, while an excellent tool, cannot support this form of system survivability.

D. CONCLUSION

Members of the HiPer-D development team saw a need to develop a real-time application emulator to help them evaluate their prototype Real-Time Distributed System

(RTDS). In order to be useful, the system would need to be able to easily emulate a wide range of real-time applications. Further, the resource usage of these emulated applications would have to be programmable. The existing set of tools available for real-time emulation did not meet their requirements.

Starting with a need and a set of requirements, the AE project set out to build an emulation application that could emulate RTDS. The main resource areas of emulation were CPU and network usage. The emulation was designed not only to match how much of a resource an application used but also to closely match when that resource was utilized. The final product, as was demonstrated through the EADSIM example, has enough built-in emulation capability and control to emulate a wide range of distributed applications accurately.

In conclusion, AE system is a tool that, in some cases, can aid developers of real-time systems. As the world becomes more dependent on computers and especially real-time computer systems for safe functionality (e.g., aircraft), the need for tools to help design and prototype future systems increases. The AE project fits nicely with the other existing tools presented in this paper and as such has the potential to aid in current and future real-time development projects.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: AE COMMAND FILE FOR EADSIM EMULATION

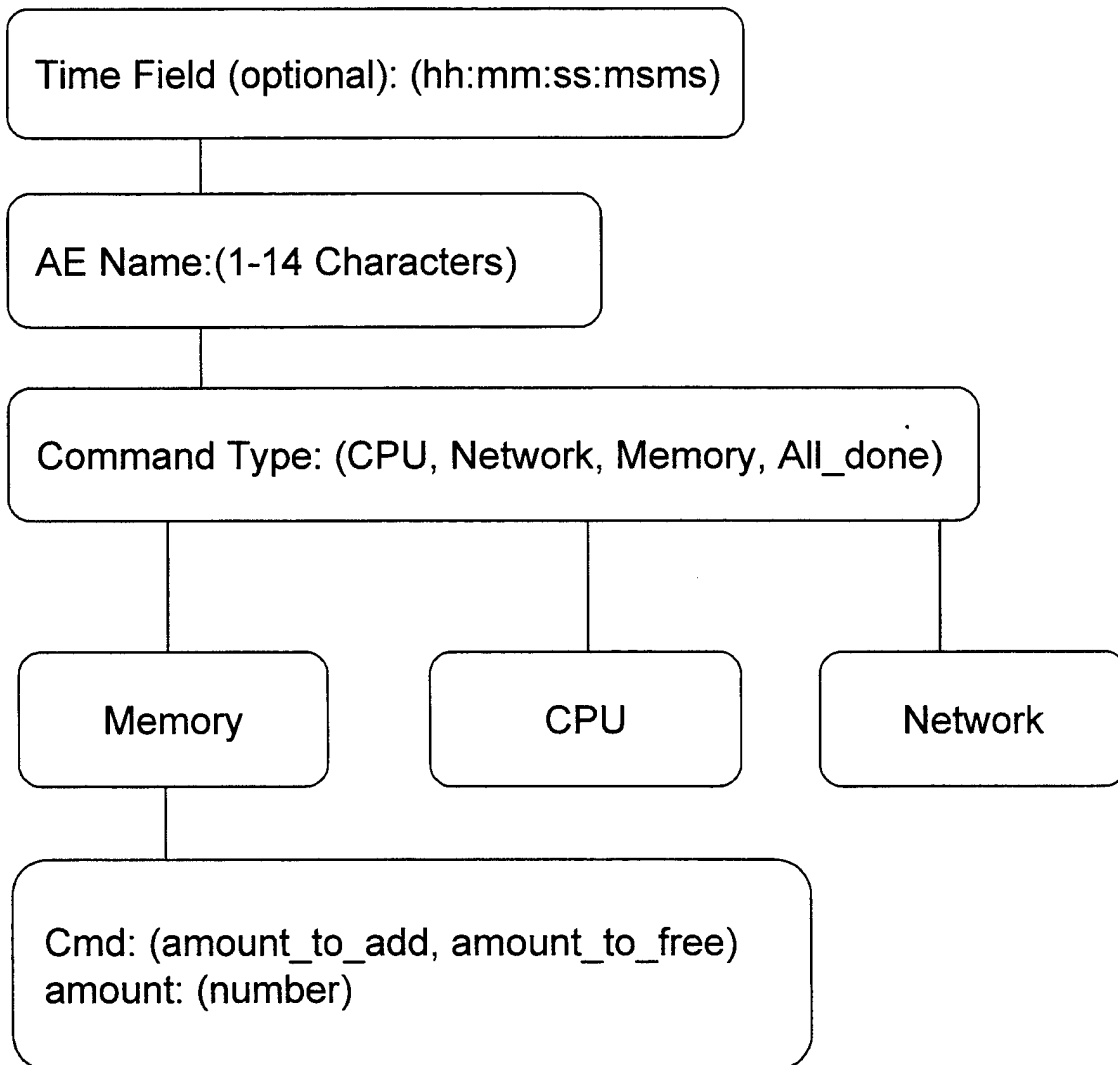
Below is the command file that was used to Emulate EADSIM. The first 10 seconds are used to create the communication channels. For example, the ":0:0:2:0:" means that 0 hours, and 0 minutes, and 2 seconds, and 0 ms after starting, execute this command. The message definitions are all separated by two seconds. While this is not necessary in theory; sometimes the AE system will have problems processing several network commands at the same time. The actual emulation process is started 10 seconds after the synchronization occurs. The "all done" and "turn off" occur well after the wall clock time for EADSIM (i.e., its CPU loaders should have executed the number of iterations programmed, recorded their CPU usage, and QoS information and exited). The "turn off" is the command for the UI to exit. At that point, the network code will report its QoS data. That last step taken is to write and close the data files. None of this is shown but is part of the normal shutdown process for an AE unit.

```
# Simple C3I :: TO :: FP and Detect
#
0:0:2:0:c3i network define_message TCP 16081 300 normal 12 1 simple fp
326 normal 15 wheat send_a_msg 3 27
c3i network define_message TCP 16082 300 normal 12 2 simple detect 356
normal 17 wheat send_a_msg 4 22
#
# Simple FP :: TO :: C3I,
#
0:0:4:0:fp network define_message TCP 16083 1389 normal 46 3 simple c3i
0 normal 0 wheat none
#
# Simple Detect :: TO :: C3I
#
0:0:6:0:detect network define_message TCP 16084 3494 normal 116 4 simple
c3i 0 normal 0 wheat none
#
0:0:10:0:c3i cpu cpu_cmd a_de 1 true 584 wheat actual 167 normal 7 17
true send_msg 2 100 160
0:0:10:200:c3i cpu cpu_cmd a_fp 1 true 584 wheat actual 167 normal 7 17
true send_msg 1 100 160
#
#
0:0:110:0:all done
0:0:112:0:turn off
```

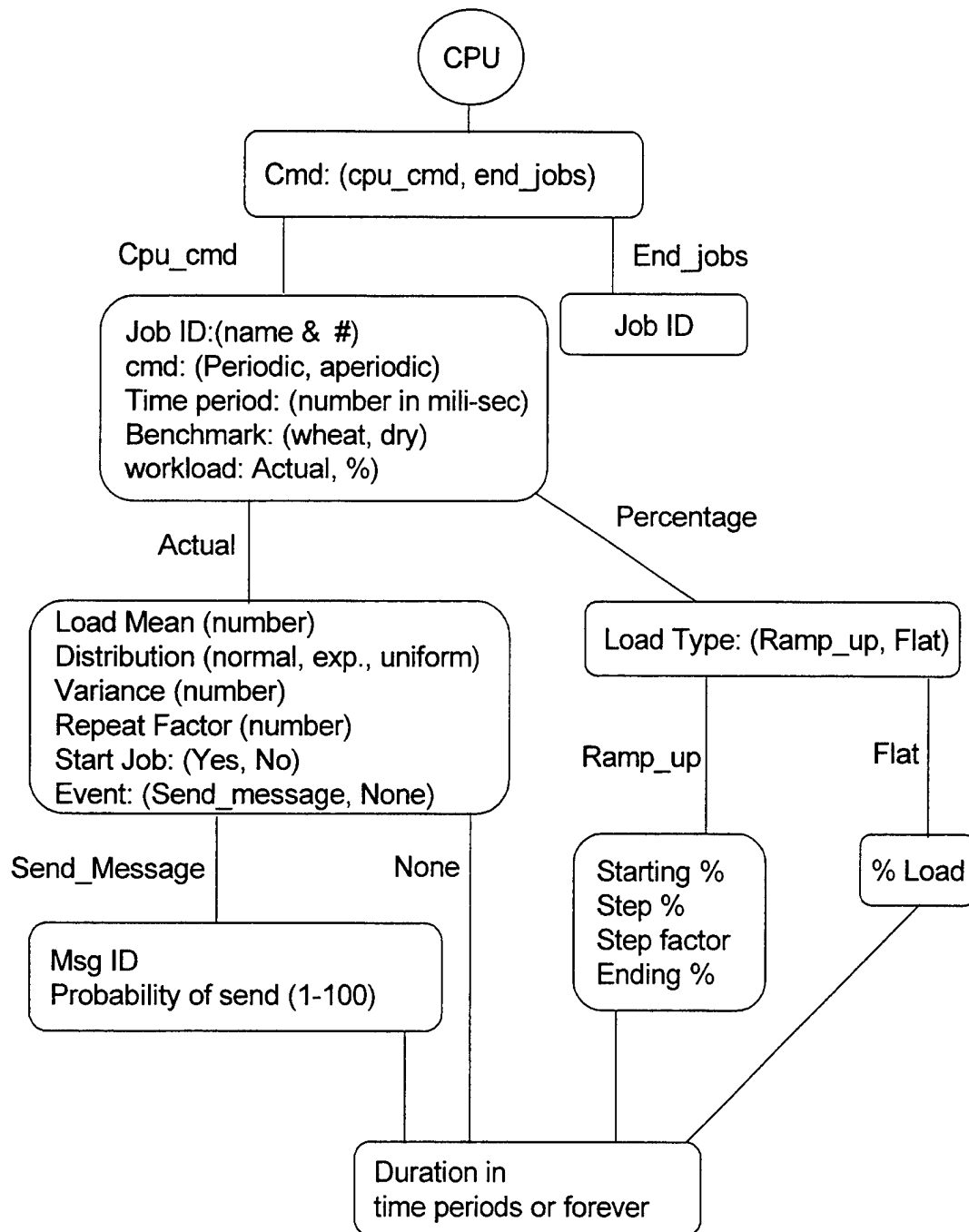
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: AE COMMAND STRUCTURE

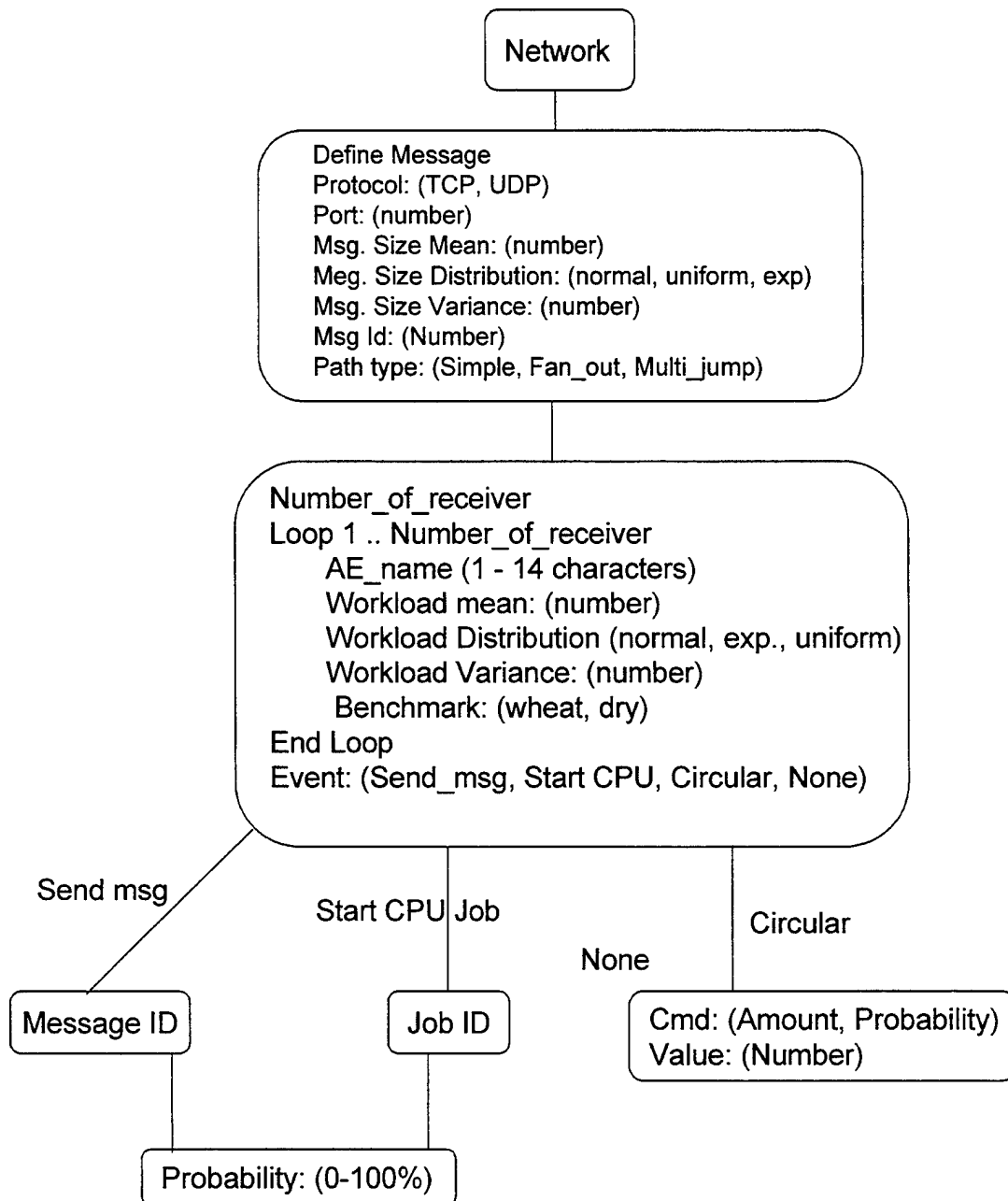
Appendix B contains the command structure for the AE system. The diagram below reads from the top to the bottom and spans the next few pages. At each location in a command, where the value in that field will cause a branch in the command the graph also has a branch and the arcs are labeled with the choices for that entry. Because some of the commands are quite long, the diagram is continued on the following pages (Network and CPU). When a valid entry for a command has only a few choices, they are placed inside parentheses.



This diagram is a continuation from the previous page and reads from top to bottom. It shows the rest of the AE CPU commands. Note, the last entry is the loader's duration in time periods (i.e., how many time periods). A loader can operate for a fixed number of time periods (e.g., 500) or, forever, as would be the case for most real-time process. A value of zero is entered when the loader should run forever.



This diagram shows the AE network command structure. Note, the number of receivers is a number from one to five. The loop in the center is where the different workload values for each receiver of a message is configured. The bottom of the diagram illustrates how events are configured. Note that None is a valid event (i.e., no event).



THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: AUTOMATED EMULATION CONFIGURATION FILES

The configuration files for the two automated startup tools are shown below. The top diagram contains the configuration of the tool that starts the other startup tool. When the AE system finishes this tool will copy the data files back to the current computer system.

The lower diagram is the configuration file for the tool that starts the AE system. It needs: the path name of the program (i.e., AE unit and UI), command line parameters and the system name where it should be run.

Load_sim_batch Example file (Eadsim batch)

```
dir
  Eadsim_results
name
  c3i alphe1
  fp alphe2
  detect alphe3
cmds
  Eadsim Eadsim.start
done
```

Start configuration file Example Eadsim. Start (DynBench tool)

```
tdrake;/home/usr/tdrake/LS;/ui.solaris2.6.exe file Eadsim.cmd 3;alphe3;
sleep 3
#
tdrake;/home/usr/tdrake/LS;/load_sim.solaris2.6.exe name:c3i ui:alphe3;alphe1;
tdrake;/home/usr/tdrake/LS;/load_sim.solaris2.6.exe name:fp ui:alphe3;alphe2;
tdrake;/home/usr/tdrake/LS;/load_sim.solaris2.6.exe name:detect ui:alphe3;alphe3;
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D: LIST OF ACRONYMS

AE	Application Emulator
AH	Hartstone Benchmark Standalone test
API	Application Programmer Interface
ASCII	American Standard Code for Information Interchange
C3I	Command, Control, Communication & Intelligence
C	The C Programming Language
C++	C "plus plus" Programming Language
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
Detect	Detection Process (part of EADSIM)
EADSIM	Extended Air Defense Simulator
FAQ	Frequently Asked Question
FP	Flight Processing (part of EADSIM)
IP	Internet Protocol
LAN	Local Area Network
MSHN	Management Systems
NSWC	Naval Surface Warfare Center
OO	Object Oriented
OS	Operating System,
PC	Personal Computer
PH	Hartstone Benchmark Standalone test (one of five defined)
PN	Hartstone Benchmark Standalone test (one of five defined)
QoS	Quality of Service
RMS	Resource Management System
RTDS	Real-Time Distributed System
SA	Hartstone Benchmark Standalone test (one of five defined)
SH	Hartstone Benchmark Standalone test (one of five defined)
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
WCS	Weapon Control System

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: DATA FROM A SERIES OF EADSIM EMULATIONS

C3i Data

Sample	Time Sec.	Work Kilo- Whetstones	Messages Sent	Messages Received	Bytes Sent	Bytes Received
1	92.96	910,744	5,440	1,363	1,719,865	3,177,358
2	92.94	910,559	5,440	1,331	1,720,289	3,131,094
3	92.93	911,995	5,440	1,339	1,718,443	3,150,189
4	92.93	908,434	5,440	1,314	1,717,910	3,086,010
5	92.93	911,405	5,440	1,378	1,718,769	3,221,250
6	92.93	909,731	5,440	1,404	1,718,382	3,267,131
7	92.93	909,406	5,440	1,385	1,719,194	3,274,533
8	92.93	911,627	5,440	1,379	1,719,091	3,240,254
9	92.93	912,939	5,440	1,365	1,719,022	3,174,493
10	92.93	911,339	5,440	1,380	1,719,926	3,273,636
11	92.93	909,341	5,440	1,388	1,718,267	3,155,483
12	92.93	910,243	5,440	1,355	1,720,314	3,153,989
13	92.93	913,600	5,440	1,397	1,718,317	3,300,024
14	92.93	908,716	5,440	1,375	1,718,660	3,185,017
15	92.94	910,333	5,440	1,377	1,718,180	3,203,098
16	92.93	912,320	5,440	1,341	1,719,018	3,115,931
17	92.93	907,315	5,440	1,385	1,717,953	3,287,455
18	92.93	908,027	5,440	1,298	1,719,756	3,054,718
19	92.93	910,554	5,440	1,377	1,719,910	3,202,327
20	92.93	906,189	5,440	1,313	1,719,558	3,033,111
21	92.93	907,806	5,440	1,349	1,719,862	3,225,880
22	92.93	912,677	5,440	1,375	1,718,270	3,236,355
23	92.93	911,514	5,440	1,317	1,721,352	3,037,212
24	92.94	909,408	5,440	1,370	1,718,171	3,241,987
25	92.93	907,627	5,440	1,357	1,719,634	3,169,059
26	92.93	904,127	5,440	1,359	1,719,005	3,187,086
27	92.94	907,137	5,440	1,360	1,718,029	3,189,261
28	92.93	909,096	5,440	1,347	1,718,775	3,186,036
29	92.93	909,624	5,440	1,348	1,716,833	3,110,805
30	92.93	910,363	5,440	1,356	1,718,549	3,159,311
31	92.92	908,086	5,440	1,306	1,716,356	3,090,926
32	92.93	908,660	5,440	1,353	1,718,992	3,191,105
33	92.92	912,916	5,440	1,376	1,718,959	3,221,792
34	92.93	908,524	5,440	1,404	1,719,739	3,329,034

35	92.94	911,536	5,440	1,391	1,718,975	3,292,900
36	92.93	911,266	5,440	1,341	1,719,292	3,181,081
37	92.93	907,960	5,440	1,299	1,717,327	3,074,857
38	92.93	911,930	5,440	1,359	1,720,580	3,128,674
39	92.93	908,068	5,440	1,339	1,718,828	3,102,043
40	92.93	907,018	5,440	1,326	1,718,019	3,104,702
41	92.93	910,926	5,440	1,375	1,718,266	3,223,316
42	92.93	908,199	5,440	1,368	1,719,412	3,108,910
43	92.93	908,613	5,440	1,357	1,717,872	3,179,651
44	92.93	912,559	5,440	1,359	1,719,178	3,176,295
45	92.93	908,074	5,440	1,447	1,720,180	3,378,578
46	92.93	907,023	5,440	1,399	1,718,609	3,323,445
47	92.93	907,188	5,440	1,377	1,719,547	3,246,892
48	92.93	906,722	5,440	1,302	1,718,584	3,067,133
49	92.93	909,927	5,440	1,379	1,719,934	3,207,577
50	92.93	911,130	5,440	1,375	1,718,935	3,258,271
51	92.93	908,042	5,440	1,398	1,720,070	3,261,265
52	92.93	910,476	5,440	1,367	1,719,873	3,202,444
53	92.93	909,012	5,440	1,314	1,719,417	3,110,609
54	92.93	905,559	5,440	1,380	1,719,466	3,185,968
55	92.93	912,566	5,440	1,349	1,719,757	3,102,280
56	92.93	908,547	5,440	1,325	1,719,433	3,051,015
57	92.93	910,164	5,440	1,361	1,718,433	3,174,327
58	92.93	909,705	5,440	1,293	1,717,365	3,063,736
59	92.93	912,955	5,440	1,364	1,718,581	3,239,674
60	92.93	909,801	5,440	1,321	1,720,910	3,126,011
61	92.92	907,551	5,440	1,366	1,719,332	3,166,106
62	92.92	910,440	5,440	1,365	1,720,295	3,177,639
63	92.93	909,346	5,440	1,376	1,717,599	3,229,892
64	92.93	909,310	5,440	1,306	1,717,819	3,078,273
65	92.93	909,300	5,440	1,330	1,717,869	3,194,821
66	92.93	912,726	5,440	1,390	1,718,936	3,276,833
67	92.93	909,184	5,440	1,350	1,718,542	3,129,802
68	92.93	910,463	5,440	1,354	1,717,450	3,233,155
69	92.93	909,643	5,440	1,367	1,718,673	3,179,600
70	92.93	906,425	5,440	1,379	1,719,138	3,239,515
71	92.93	913,111	5,440	1,367	1,718,459	3,235,067
72	92.93	908,408	5,440	1,424	1,718,922	3,364,259
73	92.93	909,967	5,440	1,402	1,717,778	3,231,122
74	92.93	908,718	5,440	1,377	1,720,109	3,269,965
75	92.93	908,376	5,440	1,358	1,720,737	3,189,299
76	92.92	909,935	5,440	1,336	1,720,902	3,132,721

77	92.93	909,516	5,440	1,325	1,720,013	3,109,189
78	92.93	912,540	5,440	1,374	1,719,753	3,203,944
79	92.93	907,754	5,440	1,399	1,719,097	3,359,174
80	92.94	908,808	5,440	1,297	1,718,432	2,939,887
81	92.93	914,022	5,440	1,411	1,718,216	3,295,320
82	92.94	908,065	5,440	1,370	1,719,103	3,137,401
83	92.93	908,598	5,440	1,376	1,718,480	3,231,753
84	92.93	912,191	5,440	1,365	1,717,798	3,202,152
85	92.93	909,791	5,440	1,362	1,720,330	3,181,126
86	92.93	907,828	5,440	1,405	1,717,959	3,293,213
87	92.93	908,430	5,440	1,378	1,721,030	3,228,356
88	92.92	908,417	5,440	1,365	1,720,160	3,197,565
89	92.93	912,816	5,440	1,378	1,719,732	3,209,248
90	92.93	911,754	5,440	1,387	1,720,489	3,324,458
91	92.93	910,501	5,440	1,358	1,719,102	3,166,563
92	92.94	912,738	5,440	1,317	1,718,719	3,020,374
93	92.92	912,995	5,440	1,421	1,719,267	3,348,046
94	92.93	912,602	5,440	1,334	1,718,098	3,120,212
95	92.93	912,207	5,440	1,347	1,719,068	3,158,549
96	92.93	913,369	5,440	1,336	1,719,704	3,035,597
97	92.92	908,862	5,440	1,351	1,719,208	3,207,424
98	92.93	909,420	5,440	1,382	1,719,438	3,248,156
99	92.93	908,125	5,440	1,362	1,719,106	3,201,921
100	92.93	907,483	5,440	1,315	1,717,599	3,059,935
101	92.93	905,470	5,440	1,342	1,719,344	3,120,826
102	92.93	913,189	5,440	1,394	1,719,186	3,235,963
103	92.93	909,965	5,440	1,383	1,718,540	3,249,418

Average	92.93	909,784	5,440	1,361	1,719,026	3,186,878
Variance	2.44E-05	4,355,968	0	919	903,875	6,876,350,770
std_dev	0	2087	0	30	951	82924

Detect

Sample	Time	Work	Message	Messages	Bytes	Bytes
	Sec.	Kilo-	s			
		Whetstones	Sent	Received	Sent	Received
1	0	967,860	609	2,720	2,139,347	816,902
2	0	969,114	609	2,720	2,139,052	817,562
3	0	968,895	612	2,720	2,149,856	815,847
4	0	969,505	597	2,720	2,099,620	816,026
5	0	967,881	623	2,720	2,183,841	815,905
6	0	969,844	623	2,720	2,191,876	815,234
7	0	967,739	643	2,720	2,255,008	816,047
8	0	969,035	632	2,720	2,213,479	815,780
9	0	967,331	609	2,720	2,133,246	815,642
10	0	966,925	643	2,720	2,260,495	816,099
11	0	968,147	587	2,720	2,054,101	816,343
12	0	968,916	605	2,720	2,125,097	816,584
13	0	968,077	641	2,720	2,256,241	815,366
14	0	969,994	607	2,720	2,131,274	815,849
15	0	969,115	614	2,720	2,152,732	814,559
16	0	968,243	598	2,720	2,093,913	815,493
17	0	968,357	648	2,720	2,276,068	815,477
18	0	969,747	595	2,720	2,089,478	816,510
19	0	967,663	616	2,720	2,156,847	816,450
20	0	968,647	575	2,720	2,016,793	815,718
21	0	968,059	642	2,720	2,253,705	815,961
22	0	968,520	631	2,720	2,213,249	815,290
23	0	969,410	573	2,720	2,013,320	817,569
24	0	969,958	634	2,720	2,231,761	815,298
25	0	968,302	609	2,720	2,139,430	815,688
26	0	969,657	617	2,720	2,167,783	815,370
27	0	967,534	615	2,720	2,162,390	814,377
28	0	967,103	624	2,720	2,195,646	817,230
29	0	969,325	589	2,720	2,066,123	814,812
30	0	968,224	604	2,720	2,123,249	815,364
31	0	969,459	605	2,720	2,125,289	814,233
32	0	967,721	624	2,720	2,188,345	816,365
33	0	968,211	622	2,720	2,184,035	815,886
34	0	967,027	653	2,720	2,296,915	816,192
35	0	968,491	648	2,720	2,269,414	815,997
36	0	967,987	625	2,720	2,196,295	816,609

37	0	968,214	604	2,720	2,118,720	814,783
38	0	967,201	592	2,720	2,075,610	816,483
39	0	967,267	589	2,720	2,067,811	816,074
40	0	969,012	599	2,720	2,103,921	815,514
41	0	969,007	623	2,720	2,187,772	816,581
42	0	967,660	574	2,720	2,014,336	815,653
43	0	969,781	617	2,720	2,162,882	815,397
44	0	968,252	613	2,720	2,153,167	816,150
45	0	967,483	649	2,720	2,280,983	816,116
46	0	967,440	655	2,720	2,301,536	816,243
47	0	968,650	634	2,720	2,224,440	815,991
48	0	969,165	599	2,720	2,100,463	815,785
49	0	968,406	614	2,720	2,156,007	816,459
50	0	966,706	639	2,720	2,246,986	815,588
51	0	967,810	626	2,720	2,198,270	816,941
52	0	967,674	617	2,720	2,170,491	816,659
53	0	969,003	610	2,720	2,141,855	815,908
54	0	967,651	605	2,720	2,119,660	816,119
55	0	966,927	586	2,720	2,051,618	815,769
56	0	967,287	577	2,720	2,022,599	815,447
57	0	968,882	610	2,720	2,140,108	815,433
58	0	967,958	603	2,720	2,116,654	814,720
59	0	967,493	639	2,720	2,244,263	815,726
60	0	968,557	613	2,720	2,152,850	816,756
61	0	968,853	606	2,720	2,121,545	816,537
62	0	967,601	607	2,720	2,133,171	816,601
63	0	967,418	623	2,720	2,194,204	814,497
64	0	968,882	598	2,720	2,103,552	815,697
65	0	968,476	642	2,720	2,250,124	815,947
66	0	966,676	639	2,720	2,243,708	815,683
67	0	967,748	597	2,720	2,093,737	815,932
68	0	967,447	641	2,720	2,251,128	815,163
69	0	968,376	609	2,720	2,135,198	816,422
70	0	969,271	630	2,720	2,208,450	815,525
71	0	968,765	635	2,720	2,226,884	816,845
72	0	970,225	659	2,720	2,313,910	816,209
73	0	969,714	610	2,720	2,140,301	815,149
74	0	968,060	645	2,720	2,263,110	817,246
75	0	969,181	618	2,720	2,169,015	816,237
76	0	967,750	604	2,720	2,125,509	816,657
77	0	968,728	603	2,720	2,116,663	816,198
78	0	968,138	617	2,720	2,161,699	816,017

79	0	969,077	673	2,720	2,362,863	815,828
80	0	968,052	541	2,720	1,898,144	815,388
81	0	969,984	635	2,720	2,227,705	814,544
82	0	967,115	586	2,720	2,059,820	816,150
83	0	968,240	626	2,720	2,199,494	815,942
84	0	968,419	621	2,720	2,178,676	815,743
85	0	966,553	611	2,720	2,147,854	815,843
86	0	967,977	636	2,720	2,237,061	815,046
87	0	968,077	625	2,720	2,194,880	816,778
88	0	968,530	618	2,720	2,171,056	816,299
89	0	967,768	616	2,720	2,161,025	816,870
90	0	968,992	665	2,720	2,332,101	817,174
91	0	968,974	605	2,720	2,129,122	815,901
92	0	969,113	569	2,720	1,991,398	816,705
93	0	968,536	655	2,720	2,296,613	815,526
94	0	967,947	602	2,720	2,111,946	815,701
95	0	969,312	610	2,720	2,143,728	816,223
96	0	967,826	561	2,720	1,968,391	816,898
97	0	969,540	634	2,720	2,221,693	816,378
98	0	968,238	630	2,720	2,212,557	816,721
99	0	969,687	623	2,720	2,186,857	816,800
100	0	966,898	587	2,720	2,058,513	814,473
101	0	968,157	597	2,720	2,097,492	814,929
102	0	968,523	619	2,720	2,169,853	815,223
103	0	968,855	628	2,720	2,209,622	815,884

Average	0.00	968,361	616	2,720	2,162,317	815,936
Variance	0	735,880	525	0	6,563,514,776	491,281
std_dev	0	858	23	0	81016	701

FP

Sample	Time	Work	Message	Message	Bytes	Bytes
			s	s		
	Sec.	Kilo-Whetstones	Sent	Received	Sent	Received
1	0	887,515	754	2,720	1,059,819	815,923
2	0	887,030	722	2,720	1,013,338	815,687
3	0	887,382	727	2,720	1,021,757	815,556
4	0	887,248	717	2,720	1,007,414	814,844
5	0	887,703	755	2,720	1,059,457	815,824
6	0	886,247	781	2,720	1,097,719	816,108
7	0	885,869	742	2,720	1,041,685	816,107
8	0	884,721	747	2,720	1,048,839	816,271
9	0	887,758	756	2,720	1,063,087	816,340
10	0	887,602	737	2,720	1,035,221	816,787
11	0	887,343	801	2,720	1,123,590	814,884
12	0	887,118	750	2,720	1,050,572	816,690
13	0	886,285	756	2,720	1,066,135	815,911
14	0	884,872	768	2,720	1,075,743	815,771
15	0	887,056	763	2,720	1,072,398	816,581
16	0	885,443	743	2,720	1,043,474	816,485
17	0	887,015	737	2,720	1,033,547	815,436
18	0	886,443	703	2,720	986,008	816,206
19	0	887,982	761	2,720	1,067,512	816,420
20	0	887,737	738	2,720	1,037,326	816,800
21	0	886,123	707	2,720	993,759	816,861
22	0	886,038	744	2,720	1,045,106	815,940
23	0	884,761	744	2,720	1,044,964	816,743
24	0	886,031	736	2,720	1,032,146	815,833
25	0	886,092	748	2,720	1,051,341	816,906
26	0	884,893	742	2,720	1,041,047	816,595
27	0	886,373	745	2,720	1,048,631	816,612
28	0	887,582	723	2,720	1,011,942	814,505
29	0	887,656	759	2,720	1,066,250	814,981
30	0	888,279	752	2,720	1,057,758	816,145
31	0	886,932	701	2,720	986,533	815,083
32	0	886,700	729	2,720	1,024,408	815,587
33	0	887,984	754	2,720	1,059,773	816,033
34	0	887,166	751	2,720	1,054,583	816,507
35	0	885,652	743	2,720	1,045,742	815,938
36	0	887,282	716	2,720	1,006,242	815,643

37	0	886,591	695	2,720	976,921	815,504
38	0	887,254	767	2,720	1,074,808	817,057
39	0	887,196	750	2,720	1,055,656	815,714
40	0	887,279	727	2,720	1,021,997	815,465
41	0	886,336	752	2,720	1,057,544	814,645
42	0	886,302	794	2,720	1,116,462	816,719
43	0	888,558	740	2,720	1,038,481	815,435
44	0	887,130	746	2,720	1,044,872	815,988
45	0	887,402	798	2,720	1,120,747	817,024
46	0	887,320	744	2,720	1,044,293	815,326
47	0	885,892	743	2,720	1,044,484	816,516
48	0	887,819	703	2,720	987,502	815,759
49	0	886,387	765	2,720	1,073,634	816,435
50	0	886,709	736	2,720	1,033,285	816,307
51	0	885,825	772	2,720	1,085,363	816,089
52	0	885,621	750	2,720	1,053,825	816,174
53	0	886,179	704	2,720	989,778	816,469
54	0	886,188	775	2,720	1,088,388	816,307
55	0	885,991	763	2,720	1,072,246	816,948
56	0	887,927	748	2,720	1,049,616	816,946
57	0	886,683	751	2,720	1,055,995	815,960
58	0	886,928	690	2,720	967,770	815,605
59	0	886,161	725	2,720	1,017,235	815,815
60	0	886,205	708	2,720	994,297	817,114
61	0	886,386	760	2,720	1,066,417	815,755
62	0	887,942	758	2,720	1,066,308	816,654
63	0	887,723	753	2,720	1,057,704	816,062
64	0	886,459	708	2,720	995,617	815,082
65	0	886,052	688	2,720	965,977	814,882
66	0	886,432	751	2,720	1,055,365	816,213
67	0	885,720	753	2,720	1,057,665	815,570
68	0	886,577	713	2,720	1,003,691	815,247
69	0	887,444	758	2,720	1,066,274	815,211
70	0	885,646	749	2,720	1,053,129	816,573
71	0	887,722	732	2,720	1,030,055	814,574
72	0	884,567	765	2,720	1,073,133	815,673
73	0	886,186	792	2,720	1,113,253	815,589
74	0	887,946	732	2,720	1,028,887	815,823
75	0	887,783	740	2,720	1,042,012	817,460
76	0	884,837	732	2,720	1,028,588	817,205
77	0	887,288	722	2,720	1,013,726	816,775
78	0	886,361	757	2,720	1,064,229	816,696

79	0	885,433	726	2,720	1,018,695	816,229
80	0	887,299	756	2,720	1,062,495	816,004
81	0	885,973	776	2,720	1,090,191	816,632
82	0	886,893	784	2,720	1,099,501	815,913
83	0	886,270	750	2,720	1,054,275	815,498
84	0	886,625	744	2,720	1,045,316	815,015
85	0	886,478	751	2,720	1,055,064	817,447
86	0	885,947	769	2,720	1,078,632	815,873
87	0	887,222	753	2,720	1,055,524	817,212
88	0	885,723	747	2,720	1,048,349	816,821
89	0	886,735	762	2,720	1,070,271	815,822
90	0	886,279	722	2,720	1,014,549	816,275
91	0	886,047	753	2,720	1,059,169	816,161
92	0	887,315	748	2,720	1,050,048	814,974
93	0	886,694	766	2,720	1,074,169	816,701
94	0	884,943	732	2,720	1,029,610	815,357
95	0	886,433	737	2,720	1,036,373	815,805
96	0	888,233	775	2,720	1,088,582	815,766
97	0	887,956	717	2,720	1,007,347	815,790
98	0	885,646	752	2,720	1,057,711	815,677
99	0	885,840	739	2,720	1,036,856	815,266
100	0	887,529	728	2,720	1,022,462	816,086
101	0	886,783	745	2,720	1,044,806	817,375
102	0	886,614	775	2,720	1,088,414	816,923
103	0	888,059	755	2,720	1,061,924	815,616

Average	0.00	886,678	745	2,720	1,046,334	816,050
Variance	0	807,717	518	0	1,018,850,600	455,702
std_dev	0	899	23	0	31919	675

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- [CARF99] Paul Carff, *Analysis on Resource Usage Information Granularity Required for Optimal Scheduling*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1999.
- [CURN76] H.J.Curnow and B.A.Wichmann: "A Synthetic Benchmark," **The Computer Journal**, 19,1 (1976), pp. 43-49 (Whetstone).
- [DHRY84] R. P.Weicker "Dhrystone: A Synthetic System Programming Benchmark", **Comm. ACM**, Vol 27 No. 10 Oct 1984, pp. 1013-1030.
- [EAD00] Teledyne Brown Engineering, "EADSIM Overview," [<http://www.eadsim.com/EADSIMBrochure.html>], Oct. 2000.
- [HART89] Nelson Weiderman "HARTSTONE: Synthetic Benchmark Requirements for Hard Real-Time Applications," Technical Report, CMU/SEI-89-TR-23 ESD-89-TR-31, Carnegie Mellon University, Pittsburgh, PA, June 1989.
- [HART90] Patrick Donohoe, Ruth Shapiro and Nelson Weillerman, "Hartstone Benchmark User's Guide, Version 1.0," Users Guide CMU/SEI-90-UG-1 ESD-90-TR-5, Carnegie Mellon University, Pittsburgh, PA, March 1990.
- [HART92] N. Weilderman and N. Kamenoff, "Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems," **The Journal of Real-Time Systems**, 1992, pp. 353-382.
- [HENS99] Debra A. Hensgen, Taylor Kidd, David St. Johns, Mathew Schnaidt, Howard Jay Siegel, Tracy D. Braun, Muthucumaru Maheswaran, Shoukat Ali, Jong-Kook Kim, Cynthia Irvine, Tim Levin, Richard F. Freund, Matt Kussow, Michael Godfrey, Alpay Duman, Paul Carff, Shirley Kidd, Viktor Prasanna, Prashanth Bhat and

- Ammar Alhusaini, "An Overview of MSHN: The Management System for Heterogeneous Networks," *Proceedings Eighth Heterogeneous Computing Workshop (HCW 99)*, San Juan, Puerto Rico, IEEE Computer Society, Los Alamitos, California, 1999.
- [LUI73] Lui C.L. and Layland J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," **Journal of the Association of Computing Machinery**, 20(1): 46-61, January 1973.
- [OBJFAQ] WWW, Comp.Object FAQ, Version: 1.0.9, Date: 4/2/96, Question 1.15: "What is the Difference Between Object-Based And Object-Oriented?," www.comp.faq.s.
- [PETE81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs N.J., 1981.
- [PORT99] N. Wayne Porter, *Resource Usage for Adaptive C4I Models in a Heterogeneous Computing Environment*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1999.
- [SCHN99] Matthew C. L. Schnaidt, *Design, Implementation, and Testing of MSHN's Application Resource Monitoring Library*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1997.
- [STEV98] W. Richard Stevens, *UNIX NETWORK PROGRAMMING*, Volume 1, Second Edition, Prentice Hall, Upper Saddle River, NJ, 1998.
- [T3] HiPer-D Group technical report, "High Performance Distributed Computing Program (HiPer-D) Engineering Testbed Three (T3) Report," Code B35, Dahlgren VA, December 31, 1998.
- [UNIX97] Unix Manual Page: `hosts.equiv(4)`, Solaris 2.7, June 23 1997.
- [WEIC90] Reinhold P. Weicker, "An Overview of Common Benchmarks," **IEEE Computer**, 23(12): pp. 65-75, December 1990.

[WELC98] Lonnie R. Welch and Michael Masters, "Towards a Taxonomy of Real-Time Mission-Critical Systems," Presented at *Real-Time Mission Critical Systems (RTMCS) Workshop*, Scottsdale, AZ, Nov. 1999.

[WELCH98] Lonnie R. Welch, Behrooz A. Shirazi, Binoy Ravindran, Charles Cavanaugh, Barath Yanamula, Russ Brucks and Eui-nam Huh, "DynBench: A Dynamic Benchmark Suite for Distributed Real-Time Systems," *IPPS/SPDP Workshop*, San Juan, Puerto Rico, pp. 1335-1349, May 1999.

THIS PAGE IS INTENTIONALLY BLANK

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center.....	2
8752 John J. Knigman Rd. STE 0944	
Ft. Belvoir, VA 22060-6218	
2. Dudley Knox Library.....	2
Naval Postgraduate School	
411 Dyer Rd.	
Monterey, CA 93943-5101	
3. Chairman, Code EC.....	1
Department of Electrical and Computer Engineering	
Naval Postgraduate School	
Monterey, CA 93943-5121	
4. Professor Cynthia Irvine, Code CS/Ic.....	2
Department of Computer Science	
Naval Postgraduate School	
Monterey, CA 93943-5193	
5. Professor Douglas J. Fouts, Code EC/Fs.....	1
Department of Electrical and Computer Engineering	
Naval Postgraduate School	
Monterey, CA 93943-5121	
6. Professor Jon Butler, Code EC/WT.....	1
Department of Electrical and Computer Engineering	
Naval Postgraduate School	
Monterey, CA 93943-5121	
7. Dr. Debra Hensgen,.....	1
OpenTV, Inc.	
401 East Middlefield Road	
Mountain View, CA 94043	
8. Michael W. Masters, Code B35.....	1
Naval Surface Warfare Center, Dahlgren Division	
Dahlgren, VA 22448-5100	
9. Robert D. Harrison, Code B35.....	1
Naval Surface Warfare Center, Dahlgren Division	
Dahlgren, VA 22448-5100	

10. Timothy Drake, Code B353
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, VA 22448-5100